N° d'ordre : 3880 ANNÉE 2010

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal et Télécommunications

École doctorale MATISSE

présentée par

Erwan Grâce

préparée à l'unité de recherche UMR6074 IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires - CAIRN
École Nationale Supérieure des Sciences Appliquées et de Technologie

---

Hiérarchie mémoire reconfigurable faible consommation pour systèmes enfouis

Thèse soutenue à Lannion
le 22 octobre 2010

devant le jury composé de :

Bernard POTTIER
Professeur à l'Université de Bretagne Occidentale
Président du jury

Smaïl NIAR
Professeur à l'Université de Valenciennes
Rapporteur

Michel PAINDAVOINE
Professeur à l'Université de Bourgogne
Rapporteur

Olivier SENTIEYS
Professeur à l'Université de Rennes 1
Directeur de thèse

Daniel CHILLET
Maître de Conférences à l'Université de Rennes 1
Co-directeur de thèse

Raphaël DAVID
Ingénieur-Chercheur au CEA List
Co-directeur de thèse

Résumé

Les progrès des technologies de la micro-électronique ont permis d'embarquer des circuits numériques dans des objets multiples et divers (téléphones, GPS, automobiles, etc.) dont ils ont enrichi les fonctionnalités et amélioré les performances à moindre coût. Conjointement, l'essor rapide et constant de ces applications a amené des contraintes de conception sans précédent (contraintes de coût, de performance, de consommation, etc.). Dans ce contexte, l'émergence des architectures reconfigurables à grain épais a ouvert la voie à de nouveaux compromis entre performances et flexibilité. À ce jour, la mise en œuvre des mécanismes de reconfiguration matérielle a principalement concerné les aspects calculatoires de ces architectures. Or, les applications embarquées (multimédia) manipulent des volumes de données croissants, engendrant une sollicitation intensive des ressources de mémorisation. En outre, l'hétérogénéité et l'évolutivité des traitements induits ne permet plus d'envisager l'élaboration de solutions de stockage dédiées dans un objectif de performance et de maîtrise de la consommation. Aussi, dans le cadre de cette thèse, nous avons développé le modèle RTL, valide et fonctionnel, d'une architecture reconfigurable que nous avons nommé MOREA (acronyme de *Memory-Oriented Reconfigurable Embedded Architecture*) et dont la structure mémoire est flexible. Celle-ci est organisée en un pavage de tuiles de traitement et de stockage qui supportent les processus d'une application. Au sein d'une tuile, les tâches du processus sont exécutées par quatre *clusters* qui intègrent des ressources mémoire et de calcul. Ces *clusters* communiquent entre eux et avec une mémoire de tuile, contenant les données partagées par les tâches du processus, grâce à une interconnexion flexible de type *crossbar*. Dès lors, cette structure permet de minimiser les mouvements de données au sein de MOREA et notamment le nombre d'accès mémoire et donc d'en atténuer l'impact sur la puissance de calcul et la dissipation énergétique du système. De plus, les gains obtenus sont maximisés grâce à une unité de génération d'adresses programmable dont l'architecture a été définie en fonction des caractéristiques des applications de traitement du signal et de l'image. Celle-ci intègre notamment un accélérateur matériel pour la génération de séquences d'adresses régulières. Cette architecture permet dès lors, comparativement à une solution programmable classique, d'améliorer significativement les performances de l'unité de génération d'adresses, d'un facteur $\times 6$ en terme de Millions d'Adresses générées Par Seconde (MAPS), tout en réduisant drastiquement sa consommation d'énergie de 96%.

Mots-clés : applications embarquées, multimédia, traitement du signal et de l'image, SoC, reconfiguration matérielle, hiérarchie mémoire, génération d'adresses, efficacité énergétique.

Abstract

The continuous improvement of digital microelectronics technologies has permitted to embed digital circuits into many different objects (e.g cell phones, GPS, cars, etc.). As a consequence, their functionality as well as their performances have been significantly enhanced at a low cost. Meanwhile, the fast and steady growth of the embedded application has brought unprecedented constraints, such as high performance and low power consumption requirements. In this context, the emergence of the coarse-grained reconfigurable architectures has led to promising trade-offs between performances and flexibility. Up to date, the hardware reconfiguration paradigm implementation has mainly affected the processing unit structure. However, embedded multimedia applications compute growing sets of data which results in a lot of memory accesses. In addition, the diversity and evolution of embedded processing do not allow to build dedicated storage unit in order to respect high performance and low power consumption requirements. So, we developped the RTL model, verified functionally and validated, of a reconfigurable architecture which has been named MOREA (acronym of Memory-Oriented Reconfigurable Embedded Architecture). MOREA is organised as an array of processing and storage tiles which can run various processes of an application. In a tile, the tasks of a process are computed by four clusters which integrate memory and computing ressources. These clusters communicate with a global memory bank, which stores the data shared by the tasks of the process, thanks to a programmable crossbar interconnection. Consequently, this structure allows to minimize data transfers within MOREA and especially the number of memory accesses and therefore, allows to reduce their impact on the computing power and energy dissipation of the system. Moreover, the resulting gains are maximized thanks to a programmable address generation unit whose architecture has been defined according to the characteristics of the digital signal and image processing applications. This unit is built from an hardware accelerator which is responsible of the generation of regular address sequences. As a result, this architecture improves significantly the performances of the address generation unit by a factor $\times 6$ *in terms of Millions of Addresses generated Per Second (MAPS) while reducing drastically its power consumption by a gain of 96%, as compared with a classic programmable solution.*

Keywords : *embedded applications, multimedia, signal and image processing, SoC, hardware reconfiguration, memory hierarchy, address generation, energy efficiency.*

Sommaire

Introduction

Contexte de l'étude

Auguste Comte, philosophe français du 19^{e} siècle, disait : « On ne connaît pas complètement une science tant qu'on n'en sait pas l'histoire. » Aussi, l'archéologie, dans sa quête de l'histoire de notre humanité, nous apprend que l'homme, dès les plus anciennes traces retrouvées de son existence, n'a eu de cesse, par la fabrication d'outils rudimentaires devenant au fil du temps de plus en plus élaborés et la mise en œuvre de techniques de plus en plus complexes, de chercher à dominer une Nature hostile en améliorant ses conditions d'existences et de travail. Ainsi, apparut l'ère des premières machines qui allèrent, de progrès en progrès, vers une automatisation de plus en plus extraordinaire. L'être humain chercha dès lors à créer des appareils doués d'intelligence, des automates à son image. Des femmes en or forgées par le dieu Héphaïstos et contées dans *L'Iliade* d'Homère (entre 850 et 750 av. J.-C.), en passant par les rouages mécaniques de la première calculatrice de Blaise Pascal (1642), jusqu'aux tubes à vide du premier ordinateur ENIAC[1] (1946) conçu pour la mise au point de la bombe atomique A américaine, chaque époque a ainsi enfanté de ces inventions prometteuses, fruit de la technologie ambiante, mais à la fiabilité bien souvent médiocre. Par exemple, l'ENIAC était sujet à une panne tous les trois jours en moyenne et il ne devait pas être arrêté durant la nuit car tout changement de température, notamment ceux intervenant lors du redémarrage, risquait de détruire ses tubes.

Après tous ces tâtonnements, l'invention du premier transistor à semi-conducteur en 1947 par trois chercheurs des Bell Telephone Laboratories, William Shockley, John Bardeen et Walter H. Brattain, a marqué le début de l'ère moderne de l'informatique, faisant migrer celle-ci de salles de plusieurs dizaines de mètres cubes à une accessibilité au grand public et améliorant notablement, outre sa fiabilité, son encombrement, sa performance (vitesse de calcul) et sa consommation d'énergie comme le démontrent les données du tableau 1. Ensuite, l'avènement de la technologie CMOS[2], avec sa consommation nulle au repos et ses procédés de fabrication simplifiés induisant une miniaturisation progressive des transistors, a permis d'embarquer des calculateurs numériques dans de multiples appareils : machines à café, téléviseurs, téléphones portables, GPS[3], automobiles, avions, etc. Aujourd'hui, les systèmes embarqués sont au cœur d'une foule d'équipements dont ils enrichissent les fonctionnalités et améliorent les performances à moindre coût, à l'image des *Smartphones* ou « téléphones intelligents » qui sont tout à la fois moyens de communication et objets

[1] ENIAC : *Electronic Numerical Integrator and Computer*
[2] CMOS : *Complementary Metal-Oxide-Semiconductor*
[3] GPS : *Global Positioning System*

multimédia, c'est-à-dire PDA[4], lecteurs audio et vidéo, consoles de jeux, navigateurs internet, etc. Par définition, un système embarqué ou enfoui est un système électronique et informatique autonome, ne possédant pas nécessairement une entrée et une sortie standards, comme un clavier ou un écran d'ordinateur, et qui est dédié à une tâche ou à un domaine de tâches bien précis.

	ENIAC	**Intel 4004**
Année	1946	1971
Complexité	18 000 tubes	2 300 transistors
Surface	160 m^2	10 mm^2
Fréquence	200 kHz	108 kHz
Performance	0.005 MOPS[5]	0.06 MOPS
Consommation	150 kW	0.3 W

Tab. 1: *Comparaison des performances du premier ordinateur ENIAC avec celles du premier microprocesseur Intel 4004. Ainsi, l'invention du transistor à semi-conducteur en 1947 a permis de réduire drastiquement la surface et la consommation des systèmes informatiques, tout en améliorant leurs performances et leur fiabilité.*

Conjointement à l'évolution rapide et constante des technologies de la microélectronique, l'essor des applications embarquées amène des contraintes de conception sans précédent. Aux traditionnelles exigences de faible coût et de performance s'ajoutent désormais celles de basse consommation énergétique et de flexibilité qui imposent la définition de nouveaux paradigmes architecturaux. L'émergence des systèmes sur silicium ou *System-on-Chip* (SoC) fait suite à la nécessité d'intégrer sur un même substrat des ressources de natures hétérogènes, chacune adaptée à une classe de traitements ou à une fonction particulière. Un SoC (figure 1) est généralement construit sur la base d'un médium de communication (par exemple un ou plusieurs bus) autour duquel gravitent différentes IP[6] : des cœurs de processeurs, des accélérateurs matériels (par exemple un CODEC[7] audio ou un décodeur vidéo), des périphériques d'entrée-sortie et de la mémoire.

Nous constatons dès lors que les applications embarquées intègrent de plus en plus de fonctionnalités, induisant d'abondants et divers types de traitements à exécuter par l'entremise possible de deux catégories d'IP, à savoir celles dédiées et celles programmables. Les solutions dédiées sont dimensionnées pour remplir une seule et unique fonction, aboutissant à un niveau de performances élevé pour la tâche demandée mais ayant l'inconvénient d'une flexibilité nulle. Cet aspect fonctionnel nécessite la conception d'autant d'IP qu'il n'y a de traitements à supporter, entraînant la mise en œuvre d'un budget silicium substantiel. D'autre part, notons que certaines fonctions sont soumises à de fréquentes évolutions en rapport avec l'émergence perpétuelle de nouvelles normes, telles que les normes de compression vidéo H.26x/MPEG[9]-x. Les nouvelles IP mises en œuvre pour ce faire entrainent des coûts de développement supplémentaires et récurrents. Les solutions programmables, quant à elles, s'appliquent à une classe de traitements relativement étendue. Elles se caractérisent

[4]PDA : *Personal Digital Assistant*
[5]MOPS : *Mega Operations Per Second*
[6]IP : *Intellectual Property*
[7]CODEC : *COder / DECoder pair*
[9]MPEG : *Motion Picture Experts Group*

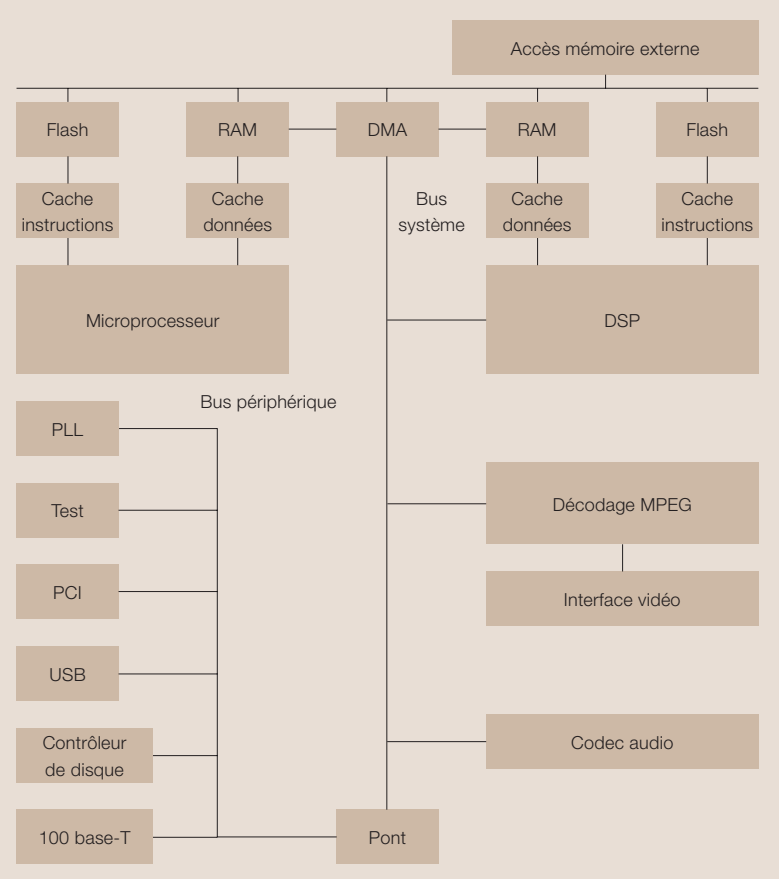


Fig. 1: *Exemple de SoC pour application embarquée multimédia. Un SoC est généralement organisé autour d'un support de communication (par exemple un ou plusieurs bus) qui assure les échangent de données entre divers modules ou IP : un microprocesseur, un DSP*[8]*, un CODEC audio, etc.*

par un degré de flexibilité important mais avec des performances dégradées. Néanmoins, ce type de structures permet de partager les ressources opératives entre plusieurs tâches et d'optimiser l'utilisation de la surface disponible. En outre, sa versatilité notoire permet l'implémentation d'algorithmes évolutifs d'un coût de développement raisonnable.

D'entre toutes ces solutions programmables, les architectures reconfigurables se parent d'un intérêt croissant dans le domaine de l'électronique embarquée. Dans la suite de cette introduction, nous allons examiner les problèmes rencontrés lors de leur conception.

Problématique de l'étude

Ces dernières années, dans un souci de performance et de maîtrise de la consommation, le bénéfice de la reconfiguration matérielle a été largement étudié au niveau de la ressource de calcul proprement dite. Le contrôle de ces architectures et leur structure de mémorisation sont restés quant à eux fondés sur des concepts conventionnels, principalement issus du paradigme Von Neumann.

Or, la conception interne d'une mémoire engendre généralement des temps de lecture et d'écriture

importants au regard des temps de calcul de l'unité de traitement. Cette divergence de performance entre les ressources de calcul et celles de stockage, ou *Memory-Processor performance Gap* (figure 2), contraint à un plafonnement de la performance globale de l'architecture jusqu'à un seuil, ou *Memory Wall*, indexé sur les temps d'accès de la mémoire.

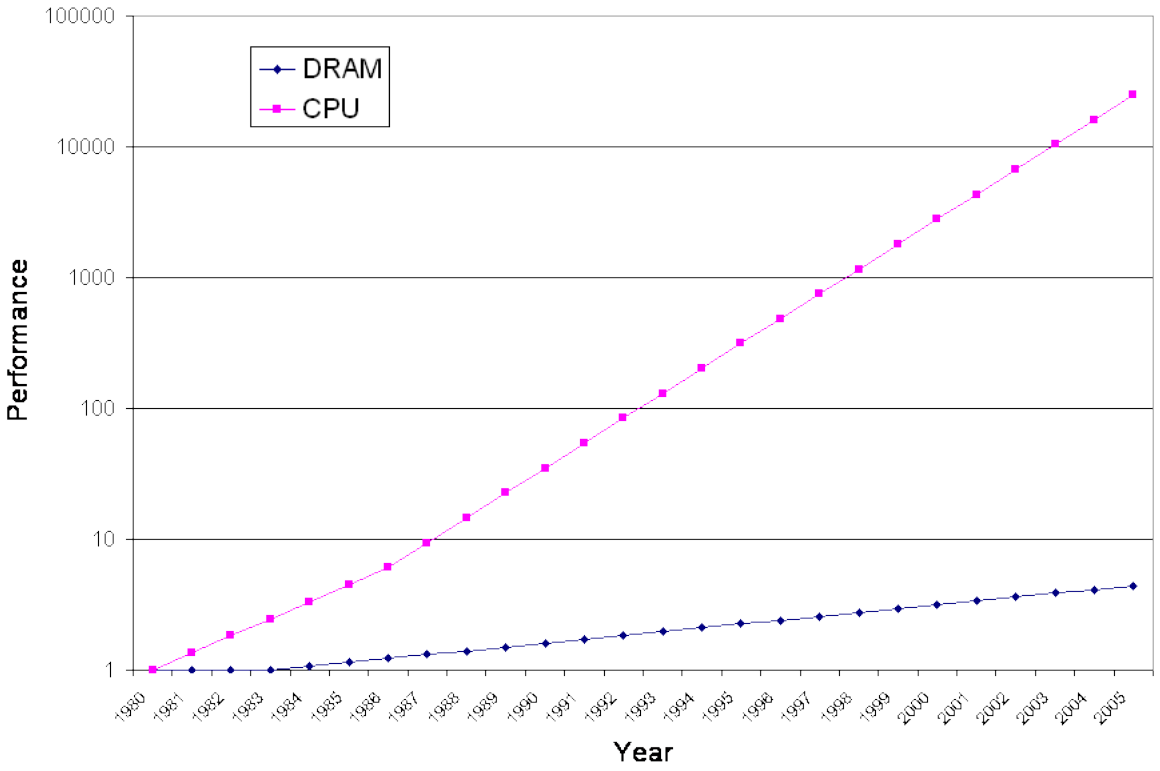


Fig. 2: *Évolution du Memory-Processor performance Gap au fil des ans (figure extraite de [1]). Depuis 1987, la différence de performance entre les ressources de calcul (CPU) et de mémorisation (DRAM) augmente de 50% chaque année.*

D'autre part, la diversification des services proposés par les applications embarquées génère une activité permanente à l'endroit de leur support d'exécution. Cette activité combinée à une augmentation de la quantité de données manipulées par ces applications, justifiée notamment par l'amélioration de la résolution des capteurs d'images [2], résulte en une sollicitation intensive de l'unité de mémorisation et, par voie de fait, d'une amplification de la consommation de celle-ci (figure 3).

Ce phénomène est en outre exacerbée par les deux facteurs suivants.

1. L'augmentation du volume de données traitées par les applications embarquées accroît les besoins en capacité de stockage des systèmes informatiques. Ainsi, les prévisions concernant la répartition de la surface des SoC nous annoncent que la mémoire s'étendra sur plus de 90% du silicium à l'aube de 2014, les 10% restant étant alloués à de la logique classique [4].
2. La réduction des dimensions des transistors les vulnérabilise vis à vis des courants de fuite, c'est-à-dire des courants formant la consommation statique, indépendante de l'activité du circuit

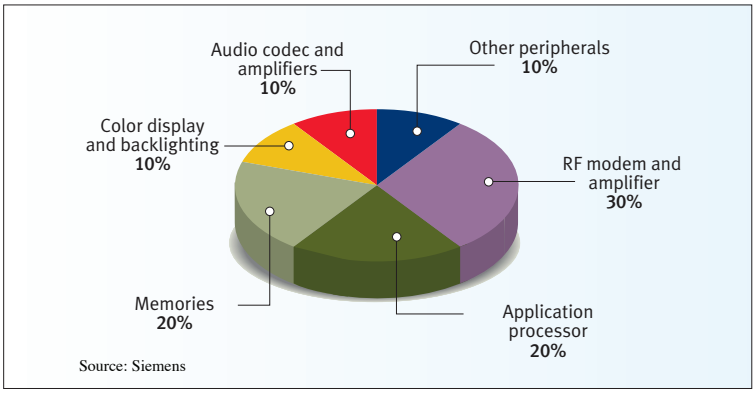


Fig. 3: *Répartition de l'énergie dissipée par une plateforme pour télécommunications mobiles (ici un Smartphone) (figure extraite de [3]). On notera sur cet exemple que les ressources mémoire représentent un cinquième de la consommation totale du composant.*

par opposition à la consommation dynamique et proportionnelle à la densité d'intégration de celui-ci (figure 4).

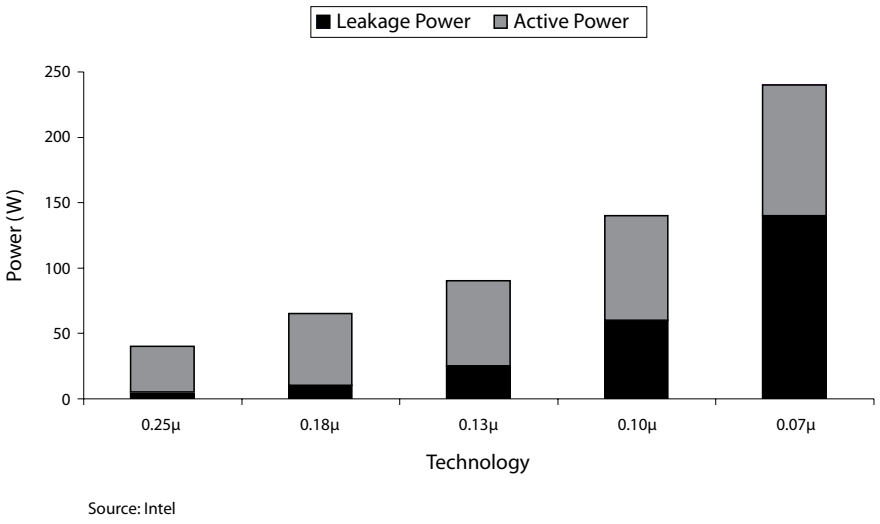


Fig. 4: *Évolution de la puissance statique consommée par un circuit intégré par rapport à la puissance dynamique en fonction de la technologie (figure extraite de [5]). La réduction des dimensions des transistors entraîne donc une augmentation de la consommation « au repos » des composants électroniques. De plus, la consommation statique, au regard de celle dynamique, tend à devenir prépondérante dans la dissipation globale des circuits intégrés.*

La conception d'une architecture reconfigurable dans un contexte embarqué implique au final de considérer équitablement la performance et la consommation, c'est-à-dire l'efficacité énergétique qui représente le niveau de performance délivrée par unité d'énergie consommée, de son unité de mémorisation. De plus, la versatilité des motifs d'accès mémoire créés par les applications de traitement du signal et de l'image exclut l'adoption de solutions dédiées pour la mise en œuvre de la partie mémoire. Aussi, dans le cadre de cette thèse, nous proposons d'étudier l'implémentation de mécanismes de reconfiguration matérielle au niveau de la structure mémoire dans un objectif de maximisation de

son efficacité énergétique. La conclusion de ca chapitre introductif expose la méthodologie de travail suivie dans cette optique.

Plan du mémoire

Le plan de ce mémoire est organisé comme suit. Tout d'abord, dans un premier chapitre, nous définirons le concept de reconfigurabilité en explorant l'espace de conception de ces architectures. Dans ce cadre, nous exprimerons notamment les forces et les faiblesses de leur unité de stockage par rapport aux besoins des fonctions de traitement de l'information. À la suite de quoi, nou établirons un état de l'art des solutions mémoire reconfigurables desquelles nous dégagerons une synthèse de leurs principaux avantages et inconvénients. Ceux-ci permettront de poser les bases de notre proposition.

Ensuite, dans un deuxième chapitre, nous décrirons notre architecture reconfigurable, que nous avons nommée MOREA (acronyme de *Memory-Oriented Reconfigurable Embedded Architecture*), dont la structure de stockage est reconfigurable. Nous introduirons, pour débuter, notre modèle d'application grâce auquel nous dégagerons l'organisation générale de MOREA. Puis, nous détaillerons son architecture en justifiant nos choix en fonction des caractéristiques de notre domaine applicatif.

Dans une troisième démarche, nous validerons notre proposition en démontrant sa capacité à exécuter les applications que nous considérons. Concrètement, nous expliquerons la mise en œuvre d'une chaîne de traitement particulière, à savoir une partie d'un encodeur vidéo H.264/MPEG-4 AVC[10]. En outre, nous profiterons de cette étape de validation pour caractériser MOREA d'un point de vue de la surface, des performances et de la consommation d'énergie. Les résultats ainsi estimés permettront de positionner notre solution dans l'espace des architectures de traitement numérique de l'information.

Nous conclurons ce mémoire en résumant notre travail et en soulignant ses contributions majeures. Nous essayerons également d'entrevoir les différentes perspectives envisageables au niveau des optimisations architecturales et de la définition d'un flot de compilation afin d'exploiter notre concept.

[10] AVC : *Advanced Video Coding*

Chapitre 1

État de l'art

1.1 Introduction au sujet des architectures reconfigurables

1.1.1 Définition générale

Qu'est-ce qu'une architecture reconfigurable ? Définir une architecture reconfigurable n'est pas chose aisée. Le concept de la reconfiguration matérielle ayant été exploré relativement récemment, et quoique présentant des similitudes avec des systèmes déjà bien connus, sa caractérisation n'en reste pas moins complexe. Néanmoins, de prime abord, la définition générale suivante, extraite de [6], peut être acceptée.

« Une architecture est dite reconfigurable dès lors qu'elle dispose d'un support lui permettant de s'adapter aux traitements qui lui sont assignés. »

L'ambiguïté des termes *support* et *traitements* dans cette définition fait de son interprétation un excercice délicat et explique la relative obscurité du concept de *reconfigurable* ainsi que la quantité de systèmes qui peuvent justifier cette dénomination. Aussi, dans le but de la préciser, la première partie de cet état de l'art permet de caractériser ces traitements tout en explorant l'espace de conception des circuits reconfigurables.

1.1.2 Positionnement des architectures reconfigurables

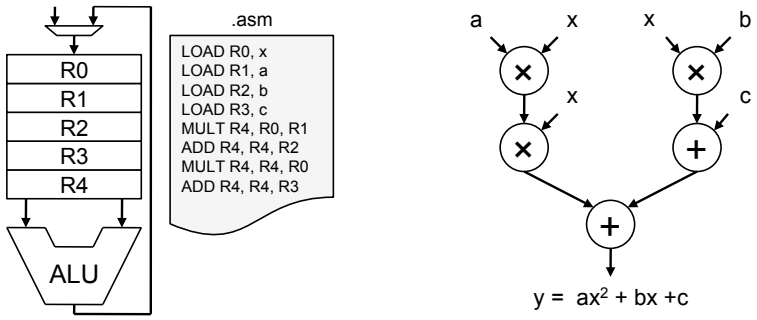
Dans le domaine du traitement du signal et de l'image, la façon dont une application est implémentée conditionne la performance et la flexibilité du support d'exécution.

Relative au paradigme Von Neumann, l'implémentation temporelle d'un traitement est fondée sur le séquencement des opérations qui sont exécutées par une unité arithmétique et logique (ALU[1], figure 1.1(a)). Un traitement est décrit par un programme (logiciel), c'est-à-dire une suite d'instructions, spécifiées dans un langage machine (assembleur), qui précise à chaque cycle machine la fonction de l'ALU et des éléments associés (file de registres, interconnexions, etc.). Il est à remarquer que cette implémentation dite temporelle peut aussi être dénommée logicielle (*software* [SW]).

[1] ALU : *Arithmetic and Logic Unit*

Le partage des ressources matérielles entre des opérations diverses et variées dénote la flexibilité de l'architecture. Toutefois, le modèle d'exécution séquentiel adopté étouffe le parallélisme de l'application et limite la puissance de calcul du matériel. Pour outrepasser cette faiblesse, dorénavant, les processeurs programmables exploitent, modestement, les types de parallélisme présents dans les applications de traitement de l'information, à savoir :

- le parallélisme d'opérations grâce aux pipelines profonds et aux processeurs VLIW[2] et superscalaires,
- le parallélisme de données à l'aide des jeux d'instructions SIMD[3] [7],
- le parallélisme de tâches par l'intermédiaire de la technique SMT[4] (processeurs superscalaires) et des puces multicœur.



(a) (b)

Fig. 1.1: *Exemple d'implémentation temporelle (a) et spatiale (b) du calcul d'un polynôme du second degré $y = ax^2 + bx + c$. Dans le premier cas de figure, les opérations à réaliser sont distribuées dans le temps et exécutées de manière séquentielle par une unité arithmétique et logique (ALU). À l'inverse, dans le deuxième cas de figure, ces mêmes opérations sont dispersées sur le silicium et assignées chacune à un opérateur dédié. Les opérateurs sélectionnés sont ensuite interconnectés grâce à un motif statique.*

A contrario, une implémentation spatiale disperse les opérations sur le silicium et les assigne chacune à un opérateur dédié. Les opérateurs sélectionnés sont interconnectés par des chemins pré-établis où s'écoulent invariablement des flots de données. La complexité de l'application est matérialisée par un agencement original de ressources opératives et de stockage. Dès lors, il est à remarquer que cette implémentation dite spatiale peut aussi être appelée matérielle (*hardware* [HW]). Sa nature flot de données optimise les temps d'exécution conjointement à la consommation énergétique, bien que la rigidité de la structure conçue annule toute flexibilité.

Une troisième voie associant une approche temporelle et spatiale définit une implémentation qualifiée de reconfigurable. Concrètement, par exemple, elle substitue aux unités spécifiques d'une implémentation matérielle des ALU interconnectées par un réseau programmable. Une application est décomposée en phases de calcul et en phases de configuration des ressources matérielles qui peuvent être *statiques* et disjointes des phases de calcul, ou bien *dynamiques* et concurrentes à l'ap-

[2]VLIW : *Very Long Instruction Word*
[3]SIMD : *Single Instruction Multiple Data*
[4]SMT : *Simultaneous MultiThreading*

plication. En définitive, les architectures reconfigurables offrent de nouvelles perspectives en terme de compromis entre la performance et la flexibilité.

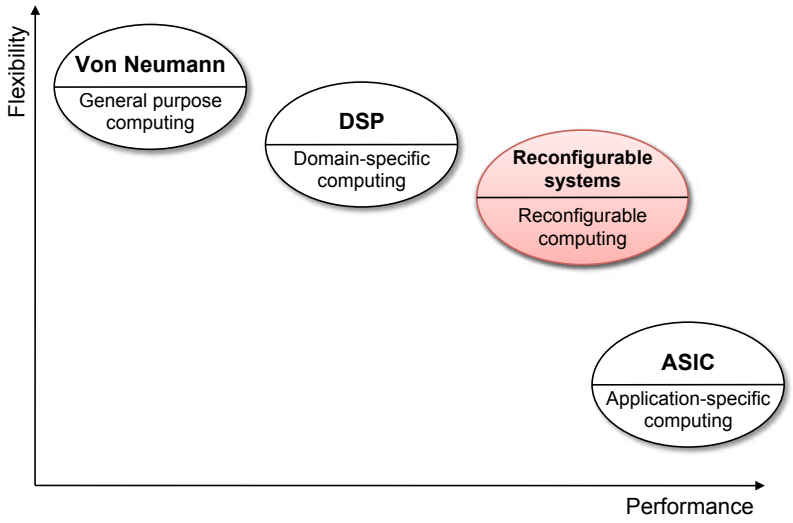


Fig. 1.2: *Positionnement relatif des architectures de traitement numérique de l'information en fonction de leur performance et de leur flexibilité (figure extraite de [8]).*

La figure 1.2 illustre le positionnement relatif des architectures de traitement numérique de l'information en fonction de leur performance et de leur flexibilité. Ainsi, leur espace de conception s'étend des processeurs programmables (Von Neumann, DSP[5]), flexibles mais peu performants, jusqu'aux circuits spécifiques (ASIC[6]), performants mais peu flexibles, en passant par les solutions reconfigurables. Mentionnons également l'existence des processeurs graphiques (GPU[7]), massivement parallèles, à la puissance de calcul prodigieuse [9], à la programmabilité croissante fondatrice du concept GPGPU[8] mais à la consommation excessive [10] excluant leur utilisation dans le secteur de l'électronique embarquée.

1.1.3 Exploration de l'espace de conception des architectures reconfigurables

Une architecture reconfigurable est structurée autour d'un réseau programmable qui interconnecte des ressources de calcul et de mémorisation. Ses performances et sa flexibilité sont essentiellement influencées par trois paramètres :

- la topologie de son réseau d'interconnexions,
- sa granularité de reconfiguration,
- la structure de son unité de stockage.

[5]DSP : *Digital Signal Processor*
[6]ASIC : *Application-Specific Integrated Circuit*
[7]GPU : *Graphics Processing Unit*
[8]GPGPU : *General-Purpose computing on GPU*

1.1.3.1 Réseau d'interconnexions

Le réseau d'interconnexions assure la distribution des données entre les ressources de calcul et mémoire. Sa structure a un impact de tout premier ordre sur la surface, les temps de propagation, la consommation et la flexibilité de l'architecture. En avant propos, trois grandes catégories de réseau peuvent être distinguées [6] :

- les réseaux globaux,
- les réseaux segmentés,
- les réseaux hiérarchisés.

Réseaux globaux Un réseau global interconnecte toutes les ressources de l'architecture. Il est constitué de bus sur lesquels se connectent les ressources grâce à des *switches* (figure 1.3). Un *switch* est un transistor qui en fonction de son état (passant ou bloqué) établit un chemin entre une entrée du réseau et une sortie. On peut donc parler d'*interrupteur* ou de *connecteur* en français. Les réseaux globaux se diversifient entre les structures *crossbar* (dans Montium [11] par exemple), extrêmement flexibles mais tout aussi avides en surface, et les structures multi-bus (dans un DPR[9] de DART [12] par exemple). Quoiqu'il en soit, ils affichent une flexibilité importante et des délais de propagation uniformes qui facilitent la tâche de placement des traitements sur des architectures centrées autour de tels motifs. Cependant, leur surface croît comme le produit du nombre d'entrées et de sorties tandis que leur temps de traversée augmente linéairement avec le nombre de sorties. Aussi, leur applicabilité se borne à des systèmes incorporant peu de ressources.

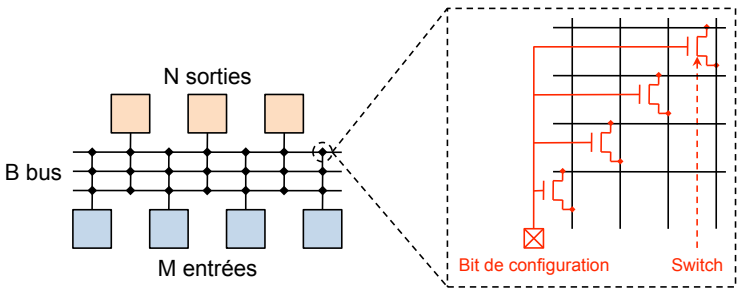


Fig. 1.3: *Structure générique d'un réseau global. Un tel réseau est composé de plusieurs bus sur lesquels se connectent les ressources grâce à des switches. Un switch est un transistor qui en fonction de son état (passant ou bloqué) établit un chemin entre une entrée du réseau et une sortie. On distingue les architectures de type crossbar, où le nombre de bus B est égal à la valeur minimale entre le nombre d'entrées M et de sorties N, et celles de type multi-bus.*

Réseaux segmentés Un réseau segmenté améliore les communications entre proches voisins au détriment de celles effectuées sur de longues distances. Dans la littérature, de nombreuses variantes ont été proposées dont les structures en anneau (dans Systolic Ring [13] par exemple) et maillées (*mesh*) (dans DReAM [14] par exemple). Elles sont fondées sur l'interconnexion de ressources contigües à des segments qui communiquent grâce à des boîtes de commutation (*Switch-Boxes*) chargées d'orienter les données dans le circuit (figure 1.4). Les transferts de données globaux devant franchir plusieurs

[9]DPR : *DataPath Reconfigurable*

SB voient leurs performances se dégrader à l'inverse des transactions locales. Toujours est-il, la performance de ce type de réseau est tributaire du partitionnement en tâches de l'application et du placement de celles-ci sur l'architecture.

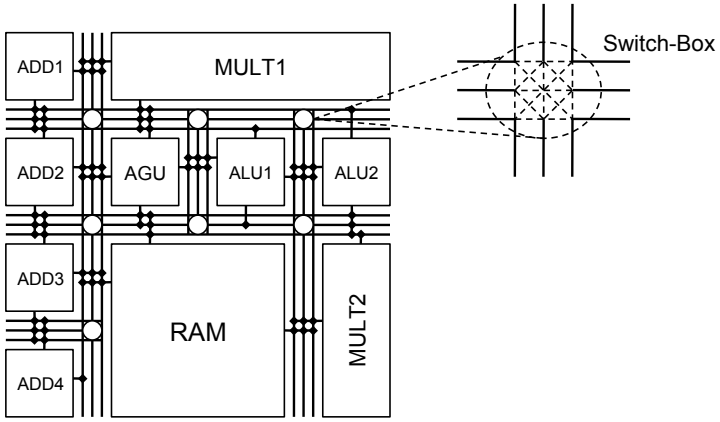


Fig. 1.4: *Exemple de réseau segmenté de type mesh généralisé. Les unités adjacentes communiquent via des segments de réseau. Ceux-ci sont interconnectés par des boîtes de commutation ou Switch-Boxes chargées d'orienter les données dans le circuit. Ainsi, les transactions globales devant franchir plusieurs SB voient leurs performances se dégrader à l'inverse des échanges locaux.*

Réseaux hiérarchisés Avec les solutions précédemment décrites, l'élévation du nombre d'unités à interconnecter affaiblissait substantiellement la performance du circuit. Pour lever cet obstacle, une alternative intéressante consiste à hiérarchiser les ressources pour en exploiter la localité des transferts de données et réduire la pénalité temporelle dûe aux communications globales (dans Pleiades [15] par exemple). Une telle structure regroupe donc les ressources au sein de *clusters* où elles communiquent par le biais d'une interconnexion globale. Au niveau supérieur, les *clusters* sont interconnectés via un réseau segmenté (figure 1.5). Dès lors, de même que précédemment, la performance du circuit dépend du partitionnement en tâches de l'application et du placement de celles-ci sur l'architecture.

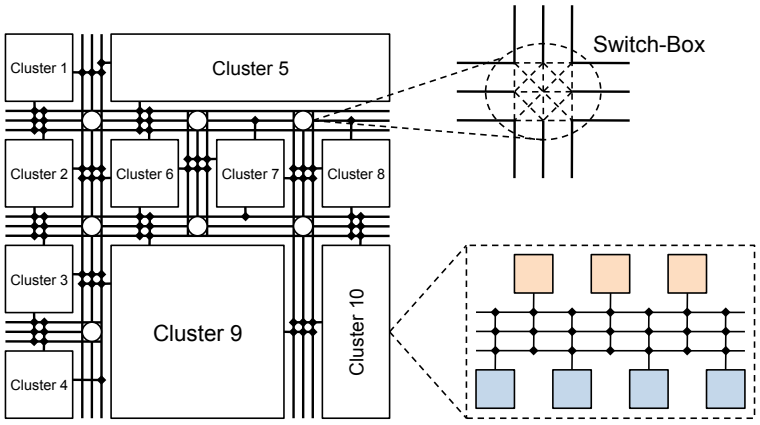


Fig. 1.5: *Exemple de réseau hiérarchisé. Dans cet exemple, les ressources sont regroupées au sein de clusters où elles communiquent grâce à une interconnexion globale. Au niveau supérieur, les clusters sont interconnectés via un réseau segmenté.*

Pour conclure, notons bien que la performance d'un réseau segmenté ou hiérarchisé dépend du partitionnement en tâches de l'application et de leur placement sur l'architecture. Ces deux opérations sont réalisables statiquement ou dynamiquement. Dans le premier cas de figure, elles peuvent intervenir à trois niveaux différents :

- au niveau algorithmique où la structure du matériel d'exécution (par exemple le nombre d'unités de calcul disponibles) influe sur la description de l'application (par exemple son découpage en procédures),
- au niveau du langage de programmation avec l'utilisation d'une bibliothèque de fonctions pré-placées par exemple,
- au niveau du compilateur, même si l'identification du parallélisme de l'application y soit relativement ardue à mettre en œuvre.

Dans le deuxième cas de figure, le placement des tâches est géré par une unité spécifique (un ordonnanceur) dont le coût temporel et surfaçique n'est pas négligeable.

1.1.3.2 Granularité de reconfiguration

La granularité de reconfiguration d'une architecture détermine son niveau de détail, c'est-à-dire la précision avec laquelle il soit possible de changer son comportement. Dans la littérature, deux épaisseurs de grain sont généralement mises en exergue :

- les architectures reconfigurables à grain fin (FPGA[10]),
- les architectures reconfigurables à grain épais (CGRA[11]).

Architectures à grain fin Les FPGA sont, pour l'essentiel, composés de plusieurs milliers de tables de scrutage (LUT[12]) interconnectées par un réseau hiérarchisé [16, 17]. Par exemple, un FPGA Xilinx Virtex-5 n'en contient pas moins de 200 000 unités. De plus, outre des LUT et des ressources de mémorisation (bascules D, blocs mémoire reconfigurables en largeur et en profondeur), ces composants intègrent des opérateurs arithmétiques (additionneurs, soustracteurs, multiplieurs) (figure 1.6), voire des cœurs de processeurs câblés pour les plus sophistiqués d'entre eux, matérialisant de fait des systèmes sur puce reconfigurables. Cependant, cette complexité matérielle implique une augmentation du nombre de boîtes de commutation au sein du réseau qui allonge significativement ses temps de propagation ainsi que sa consommation d'énergie [18]. D'autre part, elle génère un flot de données de configuration (*bitstream*) volumineux dont le chargement accapare une période de temps de plusieurs millisecondes [19].

Ainsi, la performance et la consommation d'une unité reconfigurable sont dominées par celles de son interconnexion. Par conséquent, réduire la complexité de cette dernière peut avoir un effet bénéfique sur l'efficacité énergétique du circuit. Cette stratégie est appliquée dans les architectures reconfigurables à grain épais que nous présentons dans les paragraphes suivants.

[10]FPGA : *Field-Programmable Gate Array*
[11]CGRA : *Coarse-Grained Reconfigurable Architecture*
[12]LUT : *Look-Up Table*

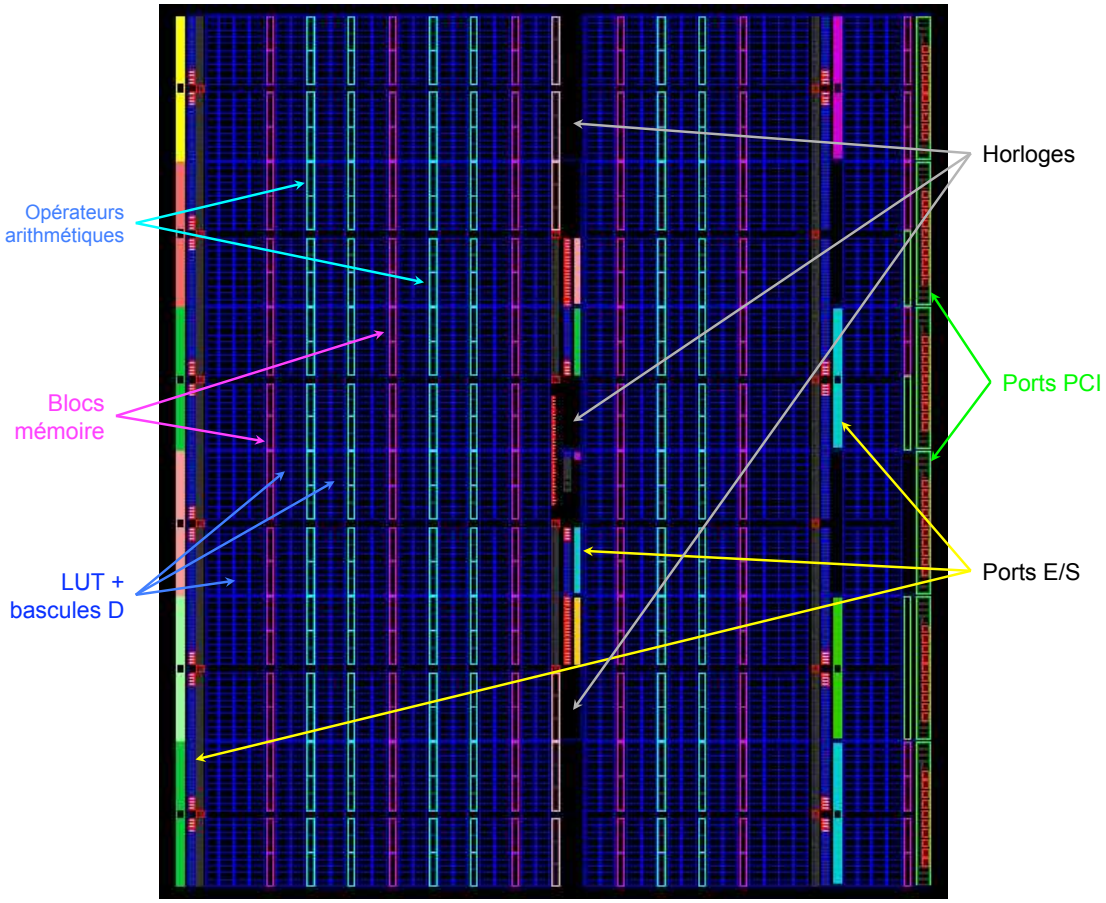


Fig. 1.6: *Vue du FPGA Xilinx Virtex-5 capturée grâce à l'outil Xilinx PlanAhead.*

Architectures à grain épais Une architecture reconfigurable à grain épais [20] substitue aux LUT travaillant sur des éléments binaires des unités travaillant sur des mots. Par exemple, la figure 2.6 montre l'architecture de DART [12].

DART est une architecture reconfigurable dynamiquement destinée aux applications de télécommunications mobiles de troisième génération. Elle est composée de plusieurs cœurs de traitement (DPR) intégrant des ressources de calcul et de stockage. Ainsi, chaque DPR dispose de quatre unités fonctionnelles (UF) à deux entrées et une sortie, dont deux multiplieurs-additionneurs (UF1 et UF3) et deux ALU (UF2 et UF4), qui réalisent des opérations sur des données 8, 16 ou 32 bits. Elles communiquent avec quatre mémoires (DM) par l'intermédiaire d'un réseau programmable de type multi-bus. D'autre part, une interconnexion segmentée permet de chaîner des opérateurs appartenant à des cœurs de traitement distincts, accroissant de ce fait leur puissance de calcul. Enfin, la reconfiguration des ressources de DART est effectuée par un contrôleur interne. Celle-ci peut être partielle et dynamique, c'est-à-dire qu'un DPR est reconfigurable indépendamment du statut (actif ou inactif) des autres cœurs de calcul, et n'excède pas quatre cycles d'horloge.

Le tableau 1.1 présente l'efficacité énergétique de trois cibles matérielles :

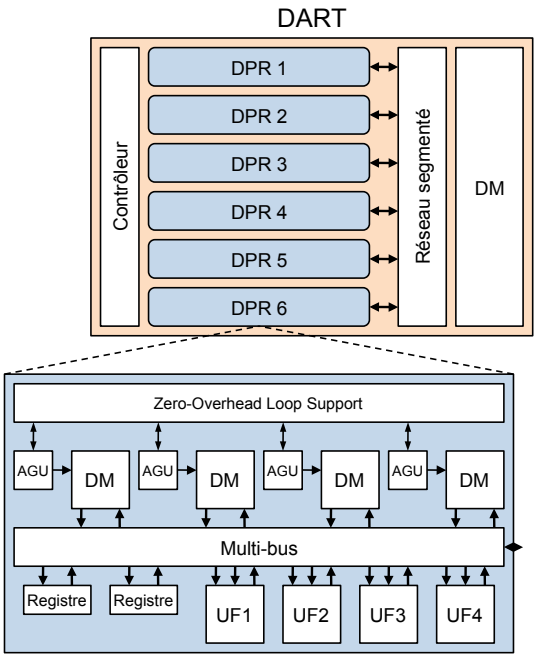


Fig. 1.7: *Architecture du processeur reconfigurable DART [12]. Elle est constituée de plusieurs cœurs de traitement (DPR) intégrant des ressources de calcul et de stockage. Plus précisément, un DPR dispose de quatre unités fonctionnelles (UF) à deux entrées et une sortie, dont deux multiplieurs-additionneurs (UF1 et UF3) et deux ALU (UF2 et UF4), qui réalisent des opérations sur des données 8, 16 ou* 32 bits. *Elles communiquent avec quatre mémoires (DM[13]) par l'entremise d'un réseau programmable de type multi-bus. Les accès aux données mémorisées sont déterminés par quatre générateurs d'adresses programmables (AGU[14]). D'autre part, une interconnexion segmentée permet de chaîner les opérateurs appartenant à des cœurs de traitement distincts, accroissant de ce fait leur puissance de calcul. Enfin, la reconfiguration dynamique des ressources de DART est effectuée par un contrôleur interne et n'excède pas quatre cycles d'horloge.*

Architecture	**Efficacité énergétique**
DART	39 MOPS/mW
FPGA Xilinx Virtex 200E	5 MOPS/mW
DSP TMS320C64x	2 MOPS/mW

Tab. 1.1: *Efficacité énergétique, exprimée en Millions d'Opérations Par Seconde et par milliWatt (MOPS/mW), d'une architecture reconfigurable à grain épais (DART), d'un FPGA et d'un processeur de traitement du signal implémentant un récepteur W-CDMA (données extraites de [12]).*

- un processeur programmable de traitement du signal (DSP TMS320C64x),
- une architecture reconfigurable à grain fin (FPGA Xilinx Virtex 200E),
- une architecture reconfigurable à grain épais (DART).

Dans le cadre de la mise en œuvre d'un récepteur W-CDMA, les données récoltées démontrent sans conteste le potentiel des mécanismes de reconfiguration matérielle, en matière d'amélioration des performances et de maîtrise de la consommation, par rapport aux approches programmables. Par ailleurs, DART délivre une efficacité énergétique supérieure d'un facteur $\times 8$ à celle d'une architecture à grain fin. Aussi, la suite de cette première partie discutera de la structure mémoire des architectures à grain épais.

1.1.3.3 Structure mémoire

Dans l'optique de répondre à la problématique que représente le *Memory-Processor performance Gap*, les ressources mémoire d'une architecture informatique sont organisées en une hiérarchie à plusieurs niveaux. Par exemple, la figure 1.8 illustre celle de MorphoSys [21].

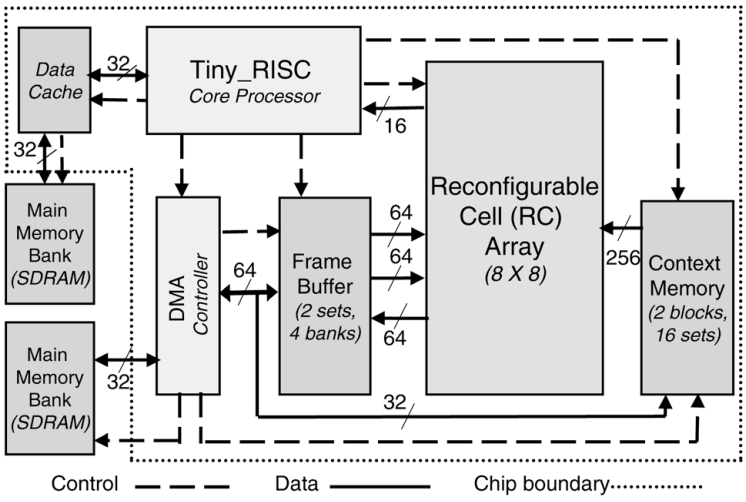


Fig. 1.8: *Architecture du système MorphoSys (figure extraite de [21]).*

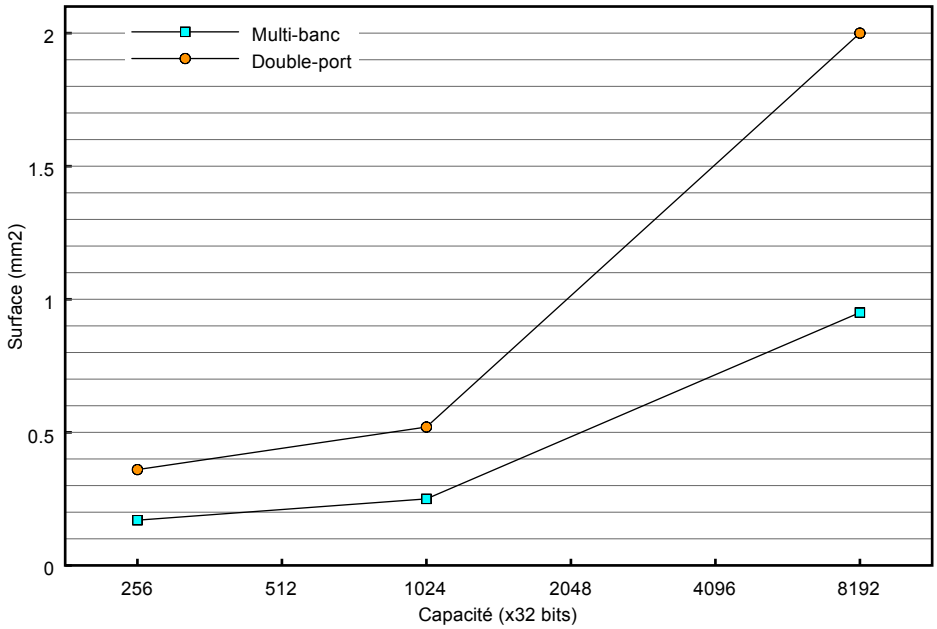
MorphoSys est construite autour d'une zone reconfigurable à gros grain (*Reconfigurable Cell Array*) dont la reconfiguration dynamique est contrôlée par un processeur programmable (*Tiny_ RISC*). L'unité reconfigurable est une matrice de 8×8 cellules interconnectées par une structure maillée et hiérarchisée. Une cellule est formée d'une ALU couplée à un multiplieur qui traitent des données 16 bits, et d'une file de quatre registres. Dans ce cadre, la mémoire de l'architecture est distribuée sur deux niveaux :

- un premier niveau (*Frame Buffer*) de faible capacité et rapide qui contient les données fréquemment accédées par les ressources de calcul,
- un deuxième niveau (*Main Memory Bank*) de grande capacité et lent qui contient toutes les données de l'application.

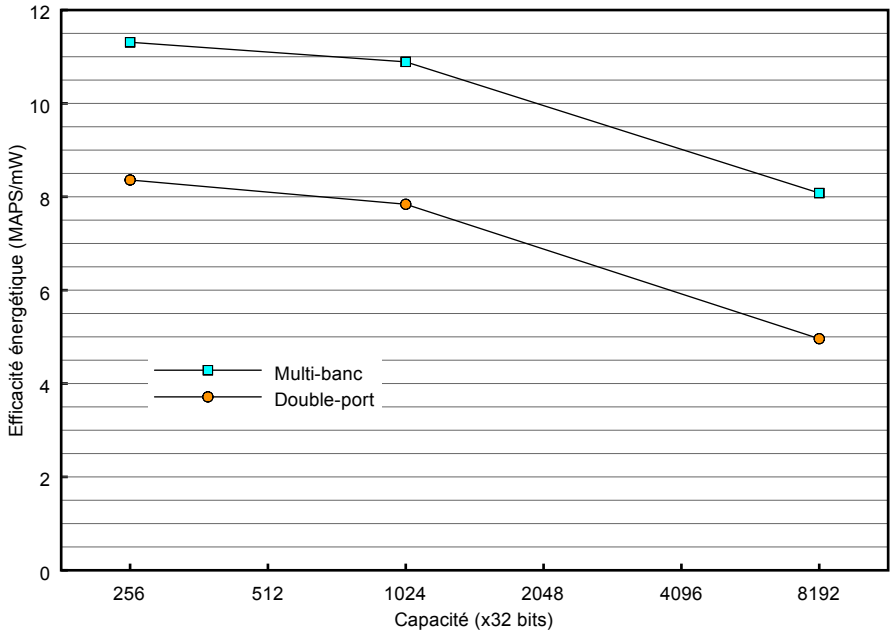
Pour approvisionner correctement les unités opératives, le premier niveau est conçu sous la forme d'une structure multi-banc, où le placement des données n'est pas transparent pour le programmeur contrairement au cas d'une structure multiport, mais dont la surface et la consommation sont moindres (figure 1.9). Par exemple, sur la base d'une technologie SRAM[15] 130 nm, une mémoire double-port de capacité 8192×32 bits consommera plus du double de silicium qu'une approche constituée de deux bancs simple-port de dimensions respectives 4096×32 bits (figure 1.1.3.3). En outre, son efficacité énergétique sera inférieure de 37.5% en moyenne à celle d'une organisation multi-banc (figure 1.1.3.3).

Ainsi, une hiérarchie mémoire n'est performante que si les accès y sont fortement localisés, c'est-à-dire si les données sont souvent réutilisées. Dans la situation inverse, les performances de l'unité

[15]SRAM : *Static Random Access Memory*



(a)



(b)

Fig. 1.9: *Comparaison en surface (a) et en efficacité énergétique (b) de mémoires double-port, c'est-à-dire disposant de deux ports de lecture/écriture, et de structures multi-banc où chaque mémoire dispose d'un seul port de lecture/écriture. Les données sont issues de la bibliothèque ST Microelectronics* 130 nm. *L'efficacité énergétique, exprimée en Millions d'Accès Par Seconde et par milliWatt (MAPS/mW), a été calculée en supposant une équiprobabilité des accès en lecture et en écriture pour chaque port mémoire.*

de stockage sont déterminées par les transferts de données entre les niveaux de la hiérarchie. Or, les applications multimédia induisent des schémas d'accès divers et variés. Par exemple, en traitement de la parole, un filtrage repose sur une opération de convolution où les données (échantillons de pa-

role et coefficients du filtre) sont lues de manière séquentielle et répétitive. À l'opposé, en traitement vidéo, une étape de quantification des valeurs d'une imagette de dimensions $N \times N$ éléments induit un parcours en zig-zag de cette imagette (figure 1.10) dont les éléments sont traités dans cet ordre.

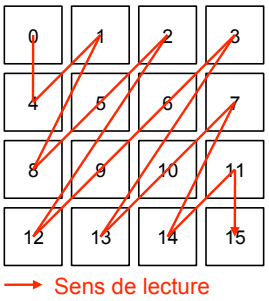


FIG. 1.10: *Illustration de la lecture d'une imagette de dimensions* 4×4 *éléments selon un parcours en zig-zag.*

D'autre part, la diversité des schémas d'accès rencontrés dans les applications de traitement de l'information engendre une diversité de connexions entre les ressources de calcul et de mémorisation de l'architecture. Pour les supporter, une structure flexible de type *crossbar* ou multi-bus est couramment implémentée. Par exemple, Montium [11] est décomposée en cinq ensembles (*Processing Parts* [PP]) qui regroupent chacun une ALU à quatre entrées et deux sorties, et deux mémoires simple-port (figure 1.11). Dans un ensemble, les communications sont mises en œuvre par une structure *crossbar*. Au niveau supérieur, les transferts de données sont assurées de manière moins flexible par une approche multi-bus, c'est-à-dire que les cinq ALU et les dix mémoires communiquement grâce à dix bus où l'absence de conflits d'accès est garantie par une unité de contrôle.

Finalement, la versatilité des applications de traitement du signal et de l'image génère des séquences d'adresses diverses et variées. Dans ce contexte, les unités de génération d'adresses (AGU) proposent généralement un mode d'adressage indexé permettant de parcourir de manière régulière des objets de type vecteur ou matrice (dans RaPiD [22] par exemple). Pour ce faire, elle comporte deux registres, stockant l'adresse courante et une valeur de pas, qui sont connectés à un additionneur pour la mise à jour de la séquence. Toutefois, dépendamment des traitements, les séquences d'adresses peuvent être plus irrégulières (par exemple la séquence *bit-reverse* [annexe A] exhibée par la FFT[16]) ou présentée des dépendances de données ou de contrôles (par exemple la dépendance de données présente dans le calcul de l'histogramme d'une image, figure 1.12). Dans ce contexte, une solution programmable doit être envisagée dans le but d'élargir le spectre des séquences d'adresses implémentables (dans DART [12] par exemple).

1.1.4 Synthèse

L'essor des applications multimédia amène des contraintes de conception sans précédent (contraintes de surface, de performance, de consommation d'énergie, de flexibilité, etc.) dans le domaine de l'électronique embarquée. Dans ce contexte, l'utilisation des architectures reconfigurables à grain

[16]FFT : *Fast Fourier Transform*

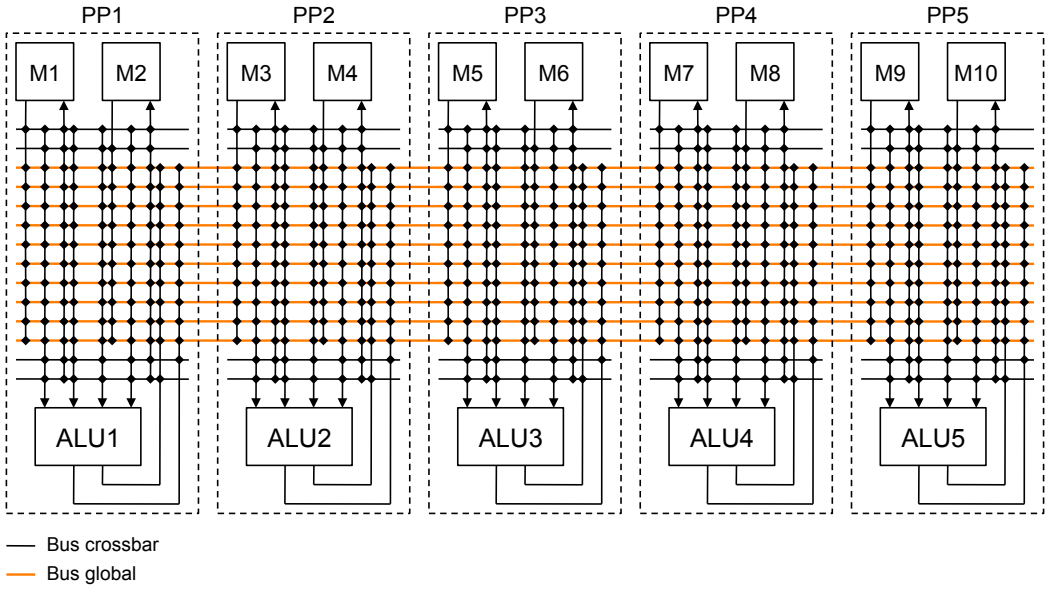


FIG. 1.11: *Architecture de Montium [11]. Elle est composée de cinq ensembles ou Processing Parts (PP) qui regroupent chacun une ALU à quatre entrées et deux sorties, et deux mémoires simple-port M_i. Dans un ensemble, les communications sont mises en œuvre par une structure crossbar. Au niveau supérieur, les transferts de données sont assurées de manière moins flexible par une approche multi-bus, c'est-à-à-dire que les cinq ALU et les dix mémoires communiquent grâce à dix bus où l'absence de conflits d'accès est garantie par une unité de contrôle.*

```
1  /* initialisation du tableau */
2  for(i=0; i<256; i++){
3     histogramme[i] = 0;
4  }
5
6  for(i=0; i<M; i++){
7     for(j=0; j<N; j++){
8        histogramme[ image[i][j] ]++; // l'indice de la case à incrémenter
9     }                                // correspond à la valeur du pixel lu
10 }
```

FIG. 1.12: *Code C du calcul de l'histogramme d'une image de dimensions $M \times N$ pixels codés sur* 8 bits.

épais ouvre de nouvelles perspectives en matière de compromis entre la surface silicium, l'efficacité énergétique et la flexibilité. Une architecture reconfigurable est construite sur la base d'un réseau programmable qui interconnecte des ressources de calcul et de stockage. Dès à présent, l'étude de la mise en œuvre des mécanismes de reconfiguration matérielle a permis d'optimiser en performance et en consommation l'interconnexion et les ressources de calcul. La mémoire, quant à elle, est généralement conçue selon une structure statique et hiérarchisée qui satisfait aux schémas d'accès mémoire fortement localisés.

Or, les applications de traitement du signal et de l'image impliquent des motifs d'accès divers et variés. D'autre part, cette versatilité impacte la structure de l'interface mémoire, c'est-à-dire l'interconnexion entre les ressources de calcul et de mémorisation, et l'unité de génération d'adresses qui

requièrent un degré de flexibilité important. Dans le cadre de nos travaux de thèse, nous envisageons d'apporter cette flexibilité, sous contrainte entre autres de performance et de consommation, via l'introduction de mécanismes de reconfiguration matérielle. La deuxième partie de cet état de l'art discutera des solutions proposées dans la littérature au sujet des structures mémoire reconfigurables.

1.2 État de l'art des structures mémoire reconfigurables

Dans les architectures reconfigurables, l'unité de stockage comprend trois parties où la reconfiguration matérielle peut être mise en œuvre :

- la hiérarchie mémoire,
- l'interface de la mémoire avec les ressources opératives,
- l'unité de génération d'adresses.

Au préalable, cette deuxième partie traite de l'implémentation de la reconfigurabilité dans les mémoires *cache*. Ces solutions sont souvent privilégiées dans les systèmes informatiques de part leur gestion exclusivement matérielle du placement des données, qui devient donc transparent pour le programmeur, conjointement à une efficacité transistor et énergétique acceptable pour bon nombre d'applications. D'autre part, dans un souci de temps d'accès minimaux, les caches sont associatifs par ensemble de plusieurs voies afin d'augmenter le temps de présence d'une donnée et réduire donc le taux de défauts d'accès. Ainsi, cette associativité implique un partitionnement du cache en plusieurs bancs qui favorise la mise en œuvre de la reconfiguration matérielle à moindre coût. Finalement, cet état de l'art s'achèvera par la présentation de trois systèmes mémoire flexibles qui représentent des solutions synthétisant les concepts abordés en matière de structure de mémorisation, d'interface mémoire et d'unité de génération d'adresses.

1.2.1 Mémoires cache reconfigurables

L'intégration de la reconfiguration matérielle dans les caches poursuit essentiellement deux objectifs distincts, à savoir l'amélioration de la performance de la mémoire [23, 24] et la réduction de sa consommation énergétique [25, 26, 27, 28, 29].

À propos de l'optimisation de la performance, Rajeev Balasubramonian [23] propose une hiérarchie reconfigurable à deux niveaux (figure 1.13). Celle-ci est partitionnée en plusieurs voies qui peuvent être affectées dynamiquement à un premier niveau *physique* ou à un deuxième niveau *virtuel*. Cette approche permet de moduler le compromis entre le temps d'un accès mémoire réussi et le taux de défauts d'accès. En outre, le temps d'accès dépendant de l'associativité du cache, c'est-à-dire de son nombre de voies, son temps de cycle est ajusté en fonction de la configuration choisie par incrément ou décrément d'un demi cycle d'horloge. Concrètement, la donnée lue est capturée par deux *latches* actifs respectivement sur chaque niveau du signal d'horloge. La reconfiguration de l'architecture est dynamique et transparente pour le programmeur. En effet, périodiquement, une routine exécutée par le processeur scrute la valeur de différents compteurs matériels disponibles dans les processeurs généralistes (par exemple le compteur du nombre de cycles consommés par instruction). À la suite de quoi, en fonction de seuils de performance définis statiquement, il est décidé d'augmenter ou pas l'associativité du premier niveau de la hiérarchie en modifiant le contenu d'un

registre de configuration. Par rapport à une structure conventionnelle à deux niveaux, l'élaboration de la reconfiguration matérielle améliore la performance du cache, exprimée en cycles par instruction, d'un gain estimé de 11%. Néanmoins, l'élévation de l'associativité du cache résultant de sa reconfiguration dégrade sa consommation d'énergie de 7% (technologie 0.1 µm).

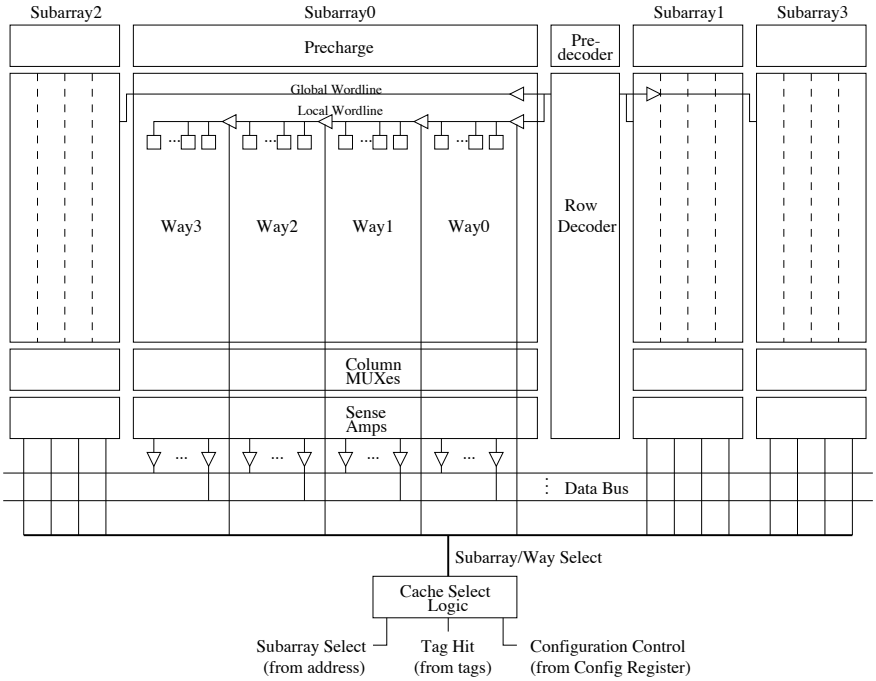


Fig. 1.13: *Exemple de hiérarchie cache reconfigurable (figure extraite de [23]). La mémoire est partitionnée en quatre bancs dont l'un d'entre eux est représenté ci-dessus. Ils sont divisés respectivement en 16 voies grâce à l'insertion de portes (buffers) trois états sur les wordlines (Global Wordlines et Local Wordlines). Dès lors, selon la configuration sélectionnée et l'adresse de la donnée désirée, certaines voies sont lues en activant les buffers adéquats.*

Dans un objectif de maîtrise de la consommation, les auteurs de [25] ont démontré le potentiel de la mise en application de la reconfiguration matérielle dans un cache. Leur proposition s'appuie sur la technique dite du *clock gating* pour inhiber le signal d'horloge au niveau d'une voie, y empêchant par conséquent toute opération de lecture et d'écriture et éliminant sa consommation dynamique (figure 1.14). Pour une structure associative par ensemble de quatre voies, sa reconfiguration dynamique est commandée grâce à l'écriture dans un registre de quatre bits (*Cache Way Select Register*) via une directive assembleur (WRCWSR : *WRite Cache Way Select Register*). Dès lors, pour cette solution, la gestion du cache, c'est-à-dire sa reconfiguration, n'est plus transparente pour le programmeur. D'autre part, dans le cas de son implantation dans un système multiprocesseur, une unité de contrôle (*Cache Controller*) reçoit des requêtes de cohérence émanant d'un deuxième niveau mémoire et permet le contrôle de l'état des données présentes dans les voies inactives. La structure ainsi mise en évidence réduit la consommation dynamique d'un gain estimé à 22% par rapport à une approche similaire et statique.

En conclusion, l'apport de la reconfiguration matérielle dans un cache engendre une amélioration

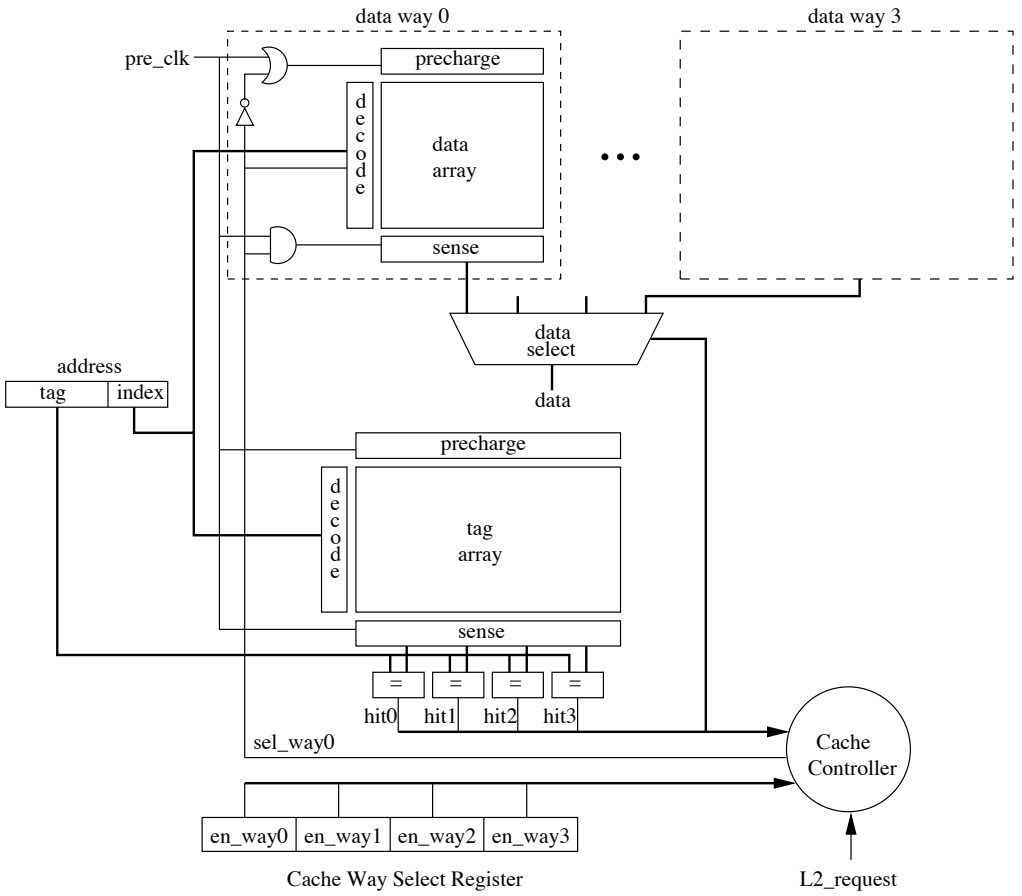


Fig. 1.14: *Exemple de mise en œuvre de la reconfiguration matérielle dans un cache dans un souci de réduction de sa consommation d'énergie (figure extraite de [25]).*

intéressante de la performance et de la consommation. Cependant, selon les cas, sa gestion n'en devient plus transparente pour le programmeur. D'autre part, à cause de politiques de remplacement de blocs généralement aléatoires, suite à un défaut d'accès, les temps de lecture dans un cache sont difficilement prédictibles. De ce fait, leur utilisation est rare dans un contexte embarqué où les contraintes temps-réel sont légions. À cela s'ajoute également la résolution des problèmes de cohérence des données dans les systèmes parallèles et qui impose le développement de mécanismes de contrôle relativement imposants (par exemple le protocole de cohérence de cache MESI[17]).

1.2.2 Interfaces mémoire

Une interface mémoire interconnecte les ressources de stockage à celles de calcul. Sa structure peut être développée de différentes façons qui dépendent de la complexité des unités opératives. Ainsi, dans un environnement multiprocesseur, les transferts de données sont asynchrones et doivent donc être synchronisés par des mécanismes de contrôle particulier. Dans le cas d'opérateurs simples (des ALU par exemple), les communications sont ordonnancées statiquement et au cycle près. La résolution des conflits d'accès au réseau est alors réalisée à la compilation. Les paragraphes suivants illustreront divers exemples adaptés aux deux contextes introduit précédemment.

[17]MESI : *Modified, Exclusive, Shared, Invalid*

1.2.2.1 Solutions pour environnement multiprocesseur

Une première approche pour le support flexible des communications dans un système multiprocesseur peut être fondée sur l'exploitation de la reconfigurabilité conséquente des architectures FPGA. Par exemple, un système pour l'analyse des images dans le domaine médical est proposé dans [30]. Celui-ci est composé de plusieurs *clusters* qui intègrent chacun six DSP interconnectés à six mémoires dynamiques par le biais d'une interconnexion reconfigurable (figure 1.15). Plus précisément, cette dernière est organisée sous la forme d'une matrice de 6×2 FPGA de la famille Xilinx XC6200. De ce fait, outre leur flexibilité autorisant l'élaboration d'une quasi infinité de topologies, la présence notamment de ressources mémoire dans les FPGA permet de matérialiser des files d'attente et, par extrapolation, de définir des protocoles de communication évolués tels que ceux implémentés dans les infrastructures NoC[18] [31].

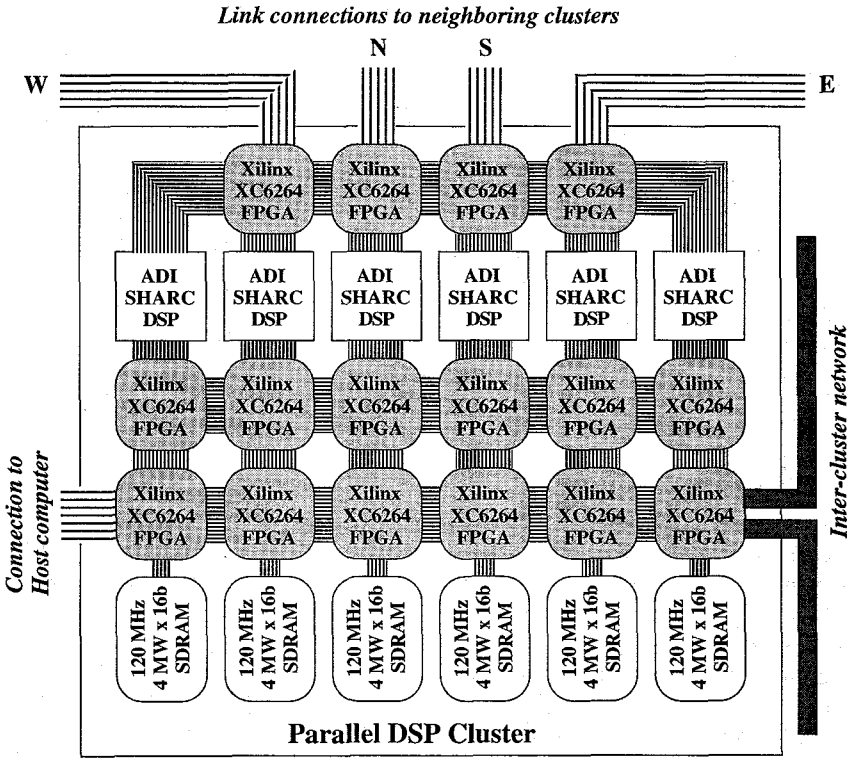


Fig. 1.15: *Exemple d'interface mémoire flexible à base de circuits FPGA pour environnement multiprocesseur (figure extraite de [30]).*

En outre, dans un contexte multimédia embarqué, une structure telle que celle décrite dans [32] peut être envisagée, avec une organisation basée sur l'implémentation d'un réseau multibus (figure 1.16). Dans ce cas de figure, chaque banc mémoire [M] est associé à un bus [B] sur lequel viennent se connecter plusieurs processeurs [P] à l'aide de *switches* [S]. Ceux-ci autorisant la connexion de plusieurs processeurs à un même bus, un arbitrage [A] est nécessaire pour résoudre les conflits d'accès pouvant s'y développer. La reconfiguration dynamique des *switches* est obtenue en un cycle par une ressource de contrôle [P_c] qui modifie le contenu d'un registre de configuration [C].

[18] NoC : *Network-on-Chip*

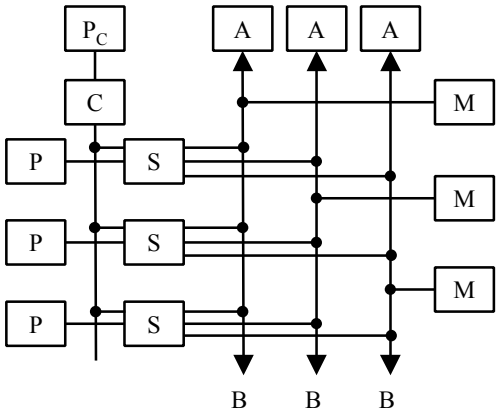


Fig. 1.16: *Exemple d'interface mémoire programmable implémentée par le biais d'un réseau multi-bus (figure extraite de [32]). Chaque mémoire [M] est associée à un bus [B] sur lequel viennent se connecter plusieurs processeurs de traitement du signal [P] grâce à des switches [S]. La configuration des switches est déterminée via un registre de configuration [C] dont le contenu est modifié par une unité de contrôle [P_c]. D'autre part, puisque plusieurs processeurs peuvent être connectés simultanément à un même bus, chaque bus dispose d'un arbitre [A] pour la résolution des conflits d'accès pouvant s'y développer.*

Enfin, les applications de traitement du signal et de l'image présentent une certaine régularité signifiée par la réalisation de calculs répétitifs. Dès lors, cette propriété entraîne une régularité des schémas de connexions élaborés entre les ressources de calcul et de mémorisation qui peuvent en conséquence être aisément prédits. Cette stratégie de contrôle de la reconfiguration de l'interface mémoire est proposée dans [33] afin de recouvrir les phases de reconfiguration de l'interconnexion et celles de transfert des données (figure 1.17). La structure de contrôle ainsi définie est constituée d'une unité de prédiction [PU[19]] adjointe à chaque processeur [P] et dont le rôle est de prédire les connexions futures entre le processeur et les mémoires [M] disponibles. Les informations émises par ces unités de prédiction sont recueillies par un élément, appelé *State Transformer*, qui anticipe les interconnexions à mettre en œuvre au niveau de l'interface mémoire. Celles-ci sont ensuite placées dans un registre à décalage cyclique, appelé *State Generator*, qui reconfigure périodiquement le réseau d'interconnexions. Dans la situation où un chemin entre une mémoire et un processeur ne serait pas disponible au moment souhaité, le processeur concerné devra transmettre explicitement une requête en ce sens à l'unité de contrôle. Bien que cette solution permette de masquer le temps de reconfiguration de l'interface, et soit relativement efficace pour des applications très régulières, elle introduit toutefois un aspect aléatoire qui n'est pas sans poser de problème vis-à-vis de contraintes temps-réel.

Dans les architectures reconfigurables, les opérateurs implémentés sont simples. Les communications qu'ils induisent sont généralement ordonnancées de manière statique et requièrent une solution d'interconnexions moins complexe que celles décrites antérieurement.

[19]PU : *Prediction Unit*

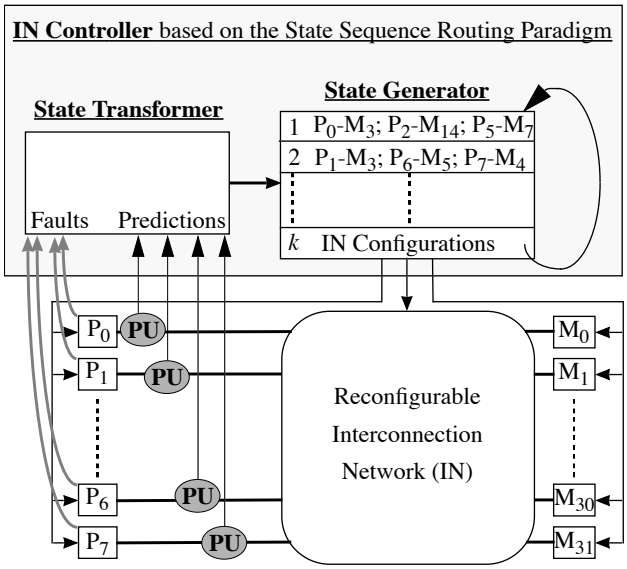


Fig. 1.17: *Exemple d'unité de contrôle pour la reconfiguration de l'interface mémoire (figure extraite de [33]).*

1.2.2.2 Solutions pour communications au sein d'une architecture reconfigurable

De manière semblable à l'architecture détaillée précédemment, les auteurs de [34] proposent une stratégie de contrôle de la reconfiguration d'une interconnexion *crossbar*. Cette manière de faire repose sur la connexion d'un opérateur i à une mémoire $(j + d)$ à l'instant t, à la condition que celui-ci soit connecté à la mémoire j à l'instant précédent $(t - 1)$. Le paramètre de configuration d est défini à partir de la représentation polyédrique d'un nid de boucles décrivant le traitement réalisé.

De surcroît, dans les applications multimédia, les tailles de données manipulées sont diverses et variées, allant de 8 bits pour le codage de la parole jusqu'à 12 voire 32 bits pour de la vision. Dans ce contexte, une interface mémoire reconfigurable peut permettre d'adapter la largeur de la mémoire dans le but de réduire sa latence et sa dissipation énergétique. Une telle structure est proposée dans [35] où une mémoire de largeur L bits est partitionnée en B bancs de largeur L/B bits. Ceux-ci sont interconnectés selon une topologie en arbre dont ils représentent les feuilles (*Memory Nodes*, figure 1.18). Les nœuds (*Root Node* et *Switch Node*, figure 1.18) sont composés d'une entrée et de deux sorties qui, en fonction de la configuration choisie, c'est-à-dire de la largeur de données sélectionnée, et des bits de poids fort de l'adresse de la donnée à lire, orientent vers leur sous-arbre gauche ou leur sous-arbre droit les signaux de contrôle de la mémoire. Ceux-ci, outre les signaux *Chip Select* et *Read/Write*, englobent également les bits de poids faible de l'adresse permettant de sélectionner une donnée dans une ou plusieurs feuilles de l'arbre.

Si cette solution optimise l'efficacité énergétique de la mémoire dans le cas d'une configuration minimale (figure 1.19(b)), en revanche, elle accroît sa surface silicium par rapport à une structure monolithique et ce, à cause notamment de la duplication des décodeurs d'adresses (figure 1.19(a)).

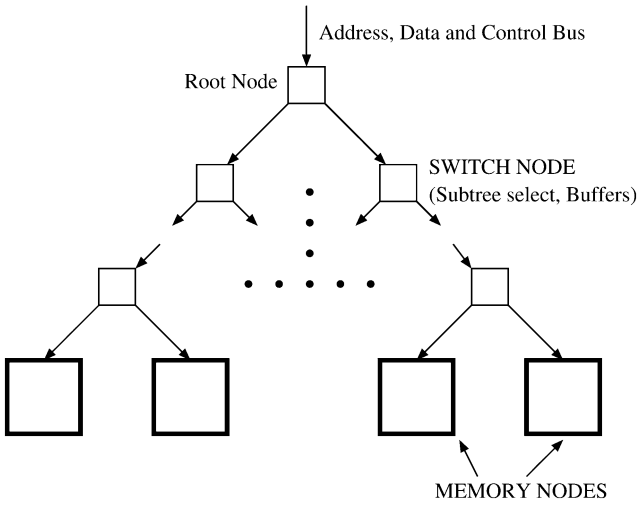


Fig. 1.18: *Exemple d'une interface mémoire programmable pour la reconfiguration de la largeur d'une structure de mémorisation (figure extraite de [36]).*

1.2.3 Architectures pour la génération d'adresses

Le problème du *Memory-Processor Gap* est imputé aux longs temps d'accès mémoire dont la définition inclut le temps de calcul de l'adresse de l'emplacement mémoire accédé. Aussi, pour lever ce verrou, une solution viable consiste à assigner le calcul des adresses à une unité *ad hoc* (AGU). Son architecture varie selon la nature des séquences d'adresses générées. De ce fait, préalablement à l'exploration de l'espace de conception des AGU, nous caractérisons les séquences d'adresses observées dans les applications multimédia.

1.2.3.1 Caractérisation des séquences d'adresses

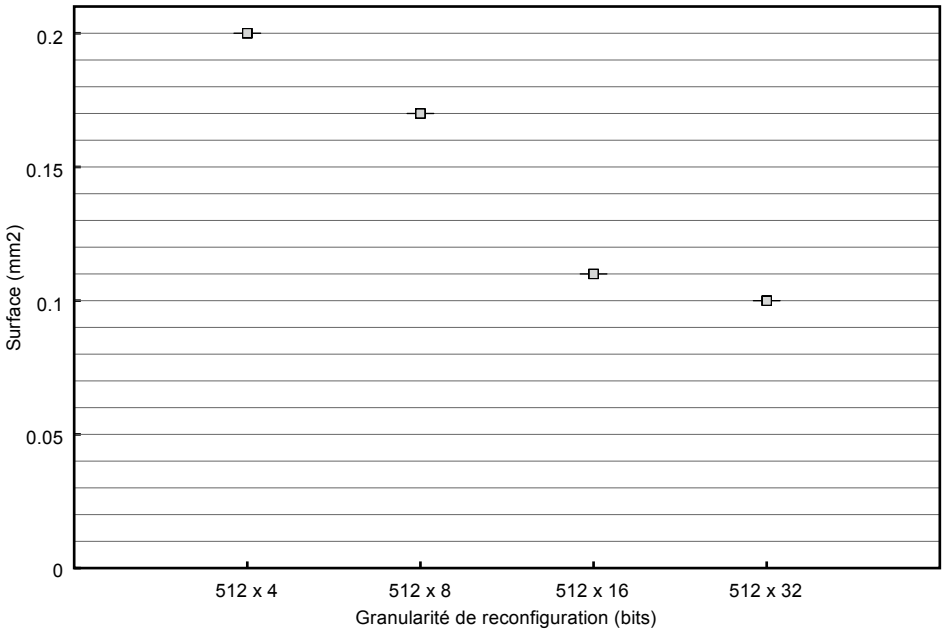
Les applications de traitement du signal et de l'image interviennent sur des données organisées en tableaux à une ou plusieurs dimensions. Ceux-ci sont manipulés par le biais de nids de boucles imbriquées dont la valeur des indices $\{i_j\}_{j=1,2,\ldots,N}$ détermine les adresses des cases du tableau à accéder. Ainsi, une séquence ou un schéma d'adresses (AE[20]) est modélisable par une fonction arithmétique ou logique de ces indices :

$$\mathrm{AE} = f(i_1, i_2, \ldots, i_N) \tag{1.1}$$

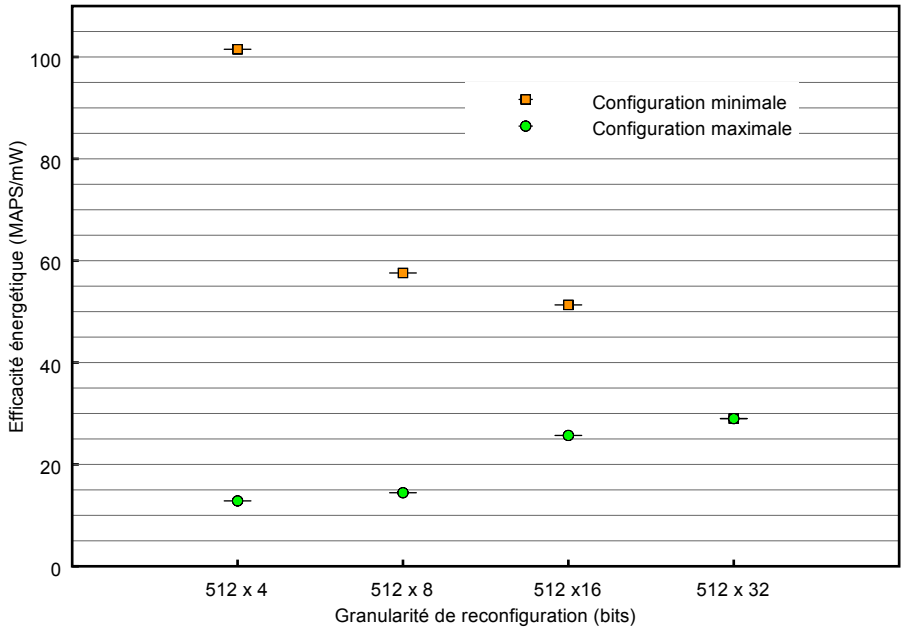
Globalement, trois types de séquences peuvent être identifiés [37] :
- les séquences affines,
- les séquences affines par morceaux,
- les séquences non linéaires.

Séquences affines Les séquences affines proviennent de nids de boucles manifestes, c'est-à-dire ne présentant pas de structures conditionnelles de type *if-then-else*. Dès lors, les schémas engendrés peuvent être représentés par une combinaison affine des indices de boucles :

[20] AE : *Address Expression*



(a)



(b)

Fig. 1.19: *Comparaison en surface (a) et en efficacité énergétique (b) d'une structure mémoire monolithique (simple-port), c'est-à-dire constituée d'un seul banc de capacité* 512×32 *bits, et de solutions reconfigurables dimensionnellement. Les données sont issues du compilateur mémoire UMC Faraday* 130 nm *et ne prennent donc pas en compte le coût d'un éventuel réseau d'interconnexions. L'efficacité énergétique, exprimée en Millions d'Accès Par Seconde et par milliWatt (MAPS/mW), suppose une équiprobabilité des opérations de lecture et d'écriture. De plus, pour chaque structure, deux valeurs sont indiquées, à savoir celle correspondant à la configuration minimale (respectivement maximale), c'est-à-dire qui repose sur une largeur de données minimale (respectivement maximale).*

$$\mathrm{AE} = C_0 + \sum_{j=1}^{N} C_j \cdot i_j \tag{1.2}$$

où $\{C_k\}_{j=1,\ldots,N}$ sont des constantes et C_0 l'adresse du premier élément accédé. Par exemple, en référence au code de la figure 1.20, nous supposons que l'adresse de l'élément a_{ij} de la matrice A est définie par la fonction g_{A} tel que :

$$g_{\mathrm{A}}(i,j) = C_0 + i \cdot N + j \tag{1.3}$$

Dans ce cas, la séquence d'adresses AE_{A} relative à la lecture du tableau A est :

$$\mathrm{AE}_{\mathrm{A}}(i,j,k) = g_{\mathrm{A}}(i,k) = C_0 + i \cdot N + k \tag{1.4}$$

```
1  for(i=0; i<N; i++){
2     for(j=0; j<N; j++){
3        C[i][j] = 0;
4        for(k=0; k<N; k++){
5           C[i][j] += A[i][k]*B[k][j];
6        }
7     }
8  }
```

Fig. 1.20: *Code C du produit matriciel $C = A \cdot B$.*

Séquences affines par morceaux Dans de nombreux traitements, les schémas d'adresses ne sont pas issus de nids de boucles manifestes. À partir de là, si les conditions des structures *if-then-else* impliquent des indices de boucles, les séquences induites peuvent être certes affines mais sur des domaines itératifs restreints. Par exemple, en se référent au code de la figure 1.21 et en supposant une fonction g_{acorr} définie tel que :

$$g_{\mathrm{acorr}}(i) = C_0 + i \tag{1.5}$$

la séquence d'adresses $\mathrm{AE}_{\mathrm{acorr}}$ correspondant à la lecture de l'objet *acorr* sera :

$$\mathrm{AE}_{\mathrm{acorr}}(m,j) = \begin{cases} g_{\mathrm{acorr}}(m) = C_0 + m, \forall m \\ g_{\mathrm{acorr}}(m-1) = C_0 - 1 + m, m > 1 \\ g_{\mathrm{acorr}}(m-2) = C_0 - 2 + m, m > 2 \\ g_{\mathrm{acorr}}(m-j) = C_0 + m - j, m > 2 \end{cases} \tag{1.6}$$

[21] LD-CELP : *Low-Delay Code Excitation Linear Prediction*
[22] ITU-T : *International Telecommunication Union Telecommunication standardization sector*

```
1  for(m=1; m<=LPC; m++){
2     K =  -acorr[m];
3     if (m>1)
4     {
5        real a1 REG(r4)=acorr[m-1];
6        c1=coeff[1];
7        tmp=c1*a1;
8        if (m>2) {
9           c1 = coeff[2]; a1 = acorr[m-2];
10          for(j=3; j<=m-1; j++) {
11             K -= tmp;
12             tmp = c1 * a1;
13             c1 = coeff[j];
14             a1 = acorr[m-j];
15          }
16          K -= tmp;
17          tmp = c1*a1;
18       }
19       K -= tmp;
20    }
21    <...>
22 }
```

Fig. 1.21: *Code C extrait de l'algorithme de compression de la parole LD-CELP*[21] *(recommandation ITU-T*[22] *G.728).*

Séquences non linéaires Les schémas ne satisfaisant pas les critères énoncés précédemment sont qualifiés de non linéaires. Cette classe de séquences d'adresses regroupe, par exemple, celles résultant du produit de deux indices de boucles ainsi que celles présentant des dépendances de données, telle la séquence engendrée par le calcul de l'histogramme d'une image (figure 1.12), ou des dépendances de contrôle.

1.2.3.2 Espace de conception des unités de génération d'adresses

De part les séquences identifiées précédemment, deux catégories de générateurs d'adresses peuvent être envisagés, à savoir ceux intégrant des compteurs matériels et ceux fondés sur des approches programmables.

Tout d'abord, le schémas affines, voire affines par morceaux, sont implémentables par l'entremise de structures reposant sur des compteurs. En effet, pour des indices de boucles dont le pas est constant, leur évolution peut être simplement matérialisée grâce à des unités de comptage, cascadées, et dont les sorties alimentent un chemin de données, flexibles ou pas, qui met en œuvre la séquence proprement dite (figure 1.22). Dans ce cas, le contrôle des différentes ressources est assuré par une machine d'états.

En revanche, pour des séquences non linéaires, entre autre, des mécanismes de contrôle avancés tels que ceux proposés dans les processeurs programmables sont requis. Dans ce contexte, les architectures de génération d'adresses développées sont basées sur des approches programmables

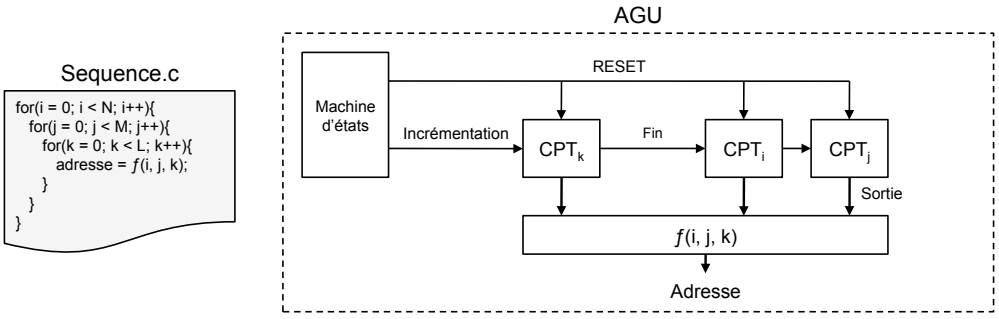


Fig. 1.22: *Exemple de générateur d'adresses à base de compteurs.*

émanant du concept RISC[23]. Par exemple, dans RMA[24] [38, 39], des AGU sont chargées de calculer les paramètres de programmation d'un contrôleur DMA[25] qui réalise les accès à une ressource de mémorisation externe au système. Concrètement, elles déterminent des descripteurs de séquences d'adresses de type *burst* (*burst descriptors*) formés de trois champs, à savoir une adresse initiale, une valeur de pas et une adresse finale. Pour ce faire, ces générateurs d'adresses sont équipés d'une mémoire d'instructions dont le décodage aboutit à la configuration d'un chemin de données centré sur une file de registres interconnectée à une ALU. Celle-ci supporte des opérations arithmétiques, dont la multiplication, et des opérations logiques (décalages, AND, etc.). De surcroît, ils disposent de mécanismes de branchements conditionnels leur permettant d'élargir le spectre des séquences d'adresses qu'ils peuvent générer.

1.2.4 Systèmes mémoire reconfigurables

Jusqu'à présent, cet état de l'art au sujet des structures mémoire reconfigurables a abordé le thème de l'implémentation de la reconfiguration matérielle au niveau de la hiérarchie mémoire, de son interconnexion avec les ressources de calcul et de son unité de génération d'adresses. Dans cette dernière partie, nous allons présenter trois sytèmes mémoire flexibles qui synthétisent, en quelque sorte, les mécanismes exposés précédemment.

1.2.4.1 RAMP : *ReconfigurAble clusters of Memory and Processor system*

RAMP [40] est une architecture multiprocesseur qui cible les applications multimédia et de télécommunications mobiles. Sa structure est hiérarchisée et comprend au plus haut niveau 16 *clusters* qui communiquent par l'intermédiaire d'un réseau comportant 16 bus de données (figure 1.23). Un *cluster* est constitué d'un processeur d'entrée et de sortie qui l'interface avec un périphérique extérieur (par exemple un capteur photos), d'une ressource mémoire et de quatre processeurs de traitement du signal et de l'image (processeur VGI[26]) qui s'échangent des données grâce à une interconnexion composée de six bus. En outre, les quatre processeurs disposent chacun de trois entrées et de trois sorties avec le réseau de niveau 1. Aussi, pour permettre notamment et autant que faire se peut les communications inter-*clusters*, des connexions de proche en proche (*bypass*) sont pourvues entre les

[23]RISC : *Reduced Instruction Set Computer*
[24]RMA : *Reconfigurable Multimedia Accelerator*
[25]DMA : *Direct Memory Access*
[26]VGI : *Vidéo, Graphics and Image processing*

différentes ressources de calcul et de stockage du *cluster*.

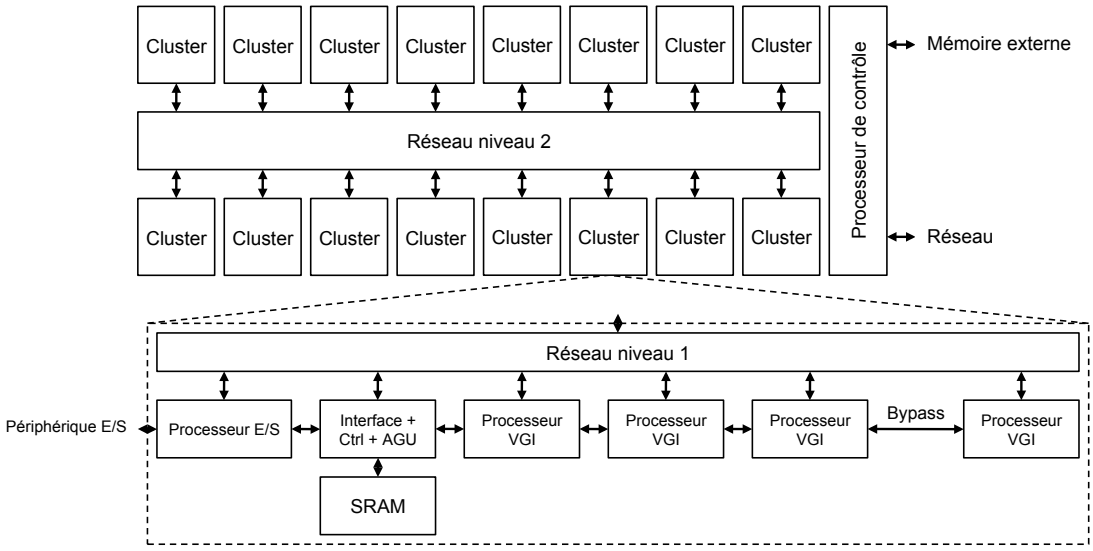


Fig. 1.23: *Architecture de RAMP.*

Globalement, les transferts de données au sein de RAMP sont asynchrones et règlementés par un protocole dit « poignée de main » (*handshake*) qui garantit, en cas de succès, des transactions dont la durée n'excède pas un cycle d'horloge. Concrètement, lorsqu'une unité émettrice souhaite transmettre une donnée, elle place celle-ci sur un bus disponible et active à 1 la ligne de contrôle associée (*handshake line*) sur le niveau bas du signal d'horloge. À partir de là, la transaction est validée si l'unité réceptrice maintient active à 1 cette ligne de contrôle sur le niveau haut suivant du signal d'horloge. Dans la situation inverse, l'unité émettrice doit répéter les étapes précédentes jusqu'au succès de la communication.

Dans un *cluster*, les données manuipulées par les quatre processeurs de traitement de l'information transitent notamment par une unité mémoire flexible [41]. Cette dernière est composée d'un banc mémoire simple-port connecté au réseau intra-*cluster* et aux interconnexions de proche en proche grâce à une interface. Celle-ci est constituée entre autre de trois multiplexeurs 2 : 1 qui permettent de sélectionner du réseau de niveau 1 une adresse de lecture, une adresse d'écriture et la donnée à écrire. Sur le port de lecture du banc mémoire est également disposé un démultiplexeur 1 : 3 qui permet de placer la donnée lue sur un bus parmi trois possibles de l'interconnexion intra-*cluster*. De plus, puisqu'une seule opération de lecture ou d'écriture ne peut être satisfaite à la fois, le chemin d'entrée de la mémoire dispose de trois files d'attente de deux registres chacune, à savoir une file pour les adresses de lecture, une file pour les adresses d'écriture et une file pour les données à écrire. Lorsqu'une file est pleine, un signal est envoyé à une unité de contrôle qui invalide ensuite toute nouvelle opération mémoire en désactivant la ligne de contrôle concernée.

D'autre part, les applications multimédia sont généralement décomposées en plusieurs tâches qui

communiquent des données grâce à des FIFO[27]. Ainsi, l'unité de contrôle implémente quatre machines d'états qui matérialisent les quatre modes de fonctionnement de la mémoire, à savoir RAM[28], FIFO, ligne à retard et LUT. Dans le mode FIFO, les adresses de lecture et d'écriture sont fournies par une AGU qui comprend pour ce faire deux compteurs incrémentaux. La taille de la FIFO est spécifiée à la reconfiguration dans un registre particulier. Selon sa valeur et celui d'un compteur bi-directionnel qui indique le nombre d'éléments présents dans la FIFO, un signal *vide* (*empty*) ou *pleine* (*full*) est envoyé au contrôleur qui refuse alors toute demande de lecture ou d'écriture. Le mode ligne à retard, quant à lui, est sensiblement identique au mode FIFO à la différence près que les opérations de lecture ne sont pas permises et sont réalisées automatiquement lorsque la ligne est pleine.

La reconfiguration de la mémoire, et plus globalement d'un *cluster*, est statique et impose la suspension de l'activité des différentes unités de calcul. Pour sa part, la reconfiguration de l'unité de stockage est régie par la mise à jour d'un registre de 55 bits.

1.2.4.2 CPMA : *Configurable Parallel Memory Architecture*

CPMA [42] est une architecture mémoire parallèle et reconfigurable qui est destinée aux applications de traitement du signal et de l'image. Elle est composée de plusieurs bancs mémoire qui sont interconnectés à des éléments de calcul grâce à une structure flexible (*Data Permutation Unit*) de type *crossbar* par exemple (figure 1.24). Ces ressources de mémorisation sont connectées à une unité de génération d'adresses qui calcule donc un ensemble de valeurs émises simultanément. Pour ce faire, elle reçoit une adresse dite virtuelle qui signifie le type de schéma d'adresses à générer et la première valeur de cette séquence. Plus précisément, cette adresse résulte de la concaténation de deux champs, à savoir le numéro d'une ligne de la table des pages et « l'adresse » du premier élément à accéder.

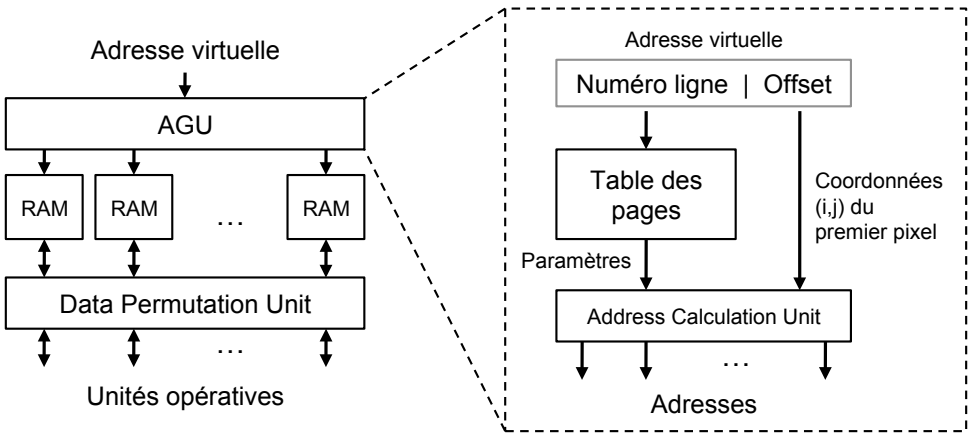


Fig. 1.24: *Organisation générale de la structure CPMA.*

La table des pages est formée de plusieurs lignes qui contiennent différents paramètres permettant

[27]FIFO : *First In First Out*
[28]RAM : *Random Access Memory*

de calculer les séquences d'adresses à générer. Par exemple, en supposant des accès à des pixels d'une image quelconque, une ligne peut contenir les informations suivantes :

- les paramètres L_i et L_j qui définissent l'étendue de la région d'intérêt de l'image selon les axes i et j définissant, respectivement, la hauteur et la largeur de l'image ;
- une fonction ayant plusieurs arguments et caractérisant le schéma d'accès à mettre en œuvre. Par exemple, la fonction $G(a_i, a_j, p)$ décrit une séquence régulière dont les valeurs a_i et a_j précisent le pas d'incrémentation selon les axes i et j, et la valeur p quantifie le nombre d'éléments à accéder.

Outre ces paramètres, il est également nécessaire de préciser dans quel banc est placé chaque pixel et à quelle adresse. Ces deux informations sont obtenues respectivement, pour un pixel de coordonnées (i, j), par les fonctions $S(i, j)$ et $a(i, j)$.

En sus de ces paramètres, le deuxième champ de l'adresse virtuelle (*offset*), indiquant dans l'exemple présent les coordonnées du pixel à accéder et situé en le plus en haut à gauche du motif, est transmis à une unité de calcul (*Address Calculation Unit*) qui détermine donc les adresses des données à lire ou à écrire.

La reconfiguration de l'architecture, et notamment le chargement d'une ou plusieurs entrées dans la table des pages, est statique et réalisée par la couche logicielle.

Ainsi, comparativement à la solution développée dans l'architecture RAMP, le générateur d'adresses construit dans CPMA offre beaucoup plus de possibilités en terme de séquences synthétisables. Toutefois, celles-ci ne peuvent, à priori, reposer sur des dépendances de données ou de contrôle. Par conséquent, ce système de mémorisation ne permet de traiter que des applications au comportement déterministe.

1.2.4.3 *Smart Memories*

Smart Memories [43] est une architecture parallèle qui est constituée au plus haut niveau d'une matrice de *quads* interconnectés par un réseau de type NoC (figure 1.25). Un *quad* comporte quatre tuiles qui communiquent grâce à un réseau flexible (réseau intra-*quad*). Une tuile est structurée autour d'un motif *crossbar* qui interface des ressources mémoire avec un processeur programmable intégrant des unités de calcul flottantes et entières et dont le chemin d'instruction peut varier d'une configuration RISC à une configuration VLIW à trois voies. D'autre part, à l'intérieur du cœur de processeur, deux unités *LOAD/STORE*, entre autre, génèrent des requêtes de lecture et d'écriture de données en direction des ressources de stockage de la tuile via les huit ports d'entrée et de sortie du processeur vers le réseau *crossbar*. Concrètement, ce dernier est décomposé en deux interconnexions, à savoir une interconnexion pour les communications du processeur vers la mémoire et une deuxième pour les communications de la mémoire vers le processeur [44]. Dans le *crossbar*, les requêtes mémoire ordonnancées statiquement sont routées dynamiquement grâce à une entête indiquant la ressource destinataire ainsi que celle émettrice. À ces informations s'ajoute un champ relatif à un mode de diffusion (*broadcast*) permettant d'envoyer une requête vers plusieurs unités de stockage. Ainsi, de part l'ordonnancement statique des requêtes, cette interface mémoire ne nécessite pas d'unités d'ar-

bitrage et, de part leur routage dynamique, ne requiert pas de phases de reconfiguration.

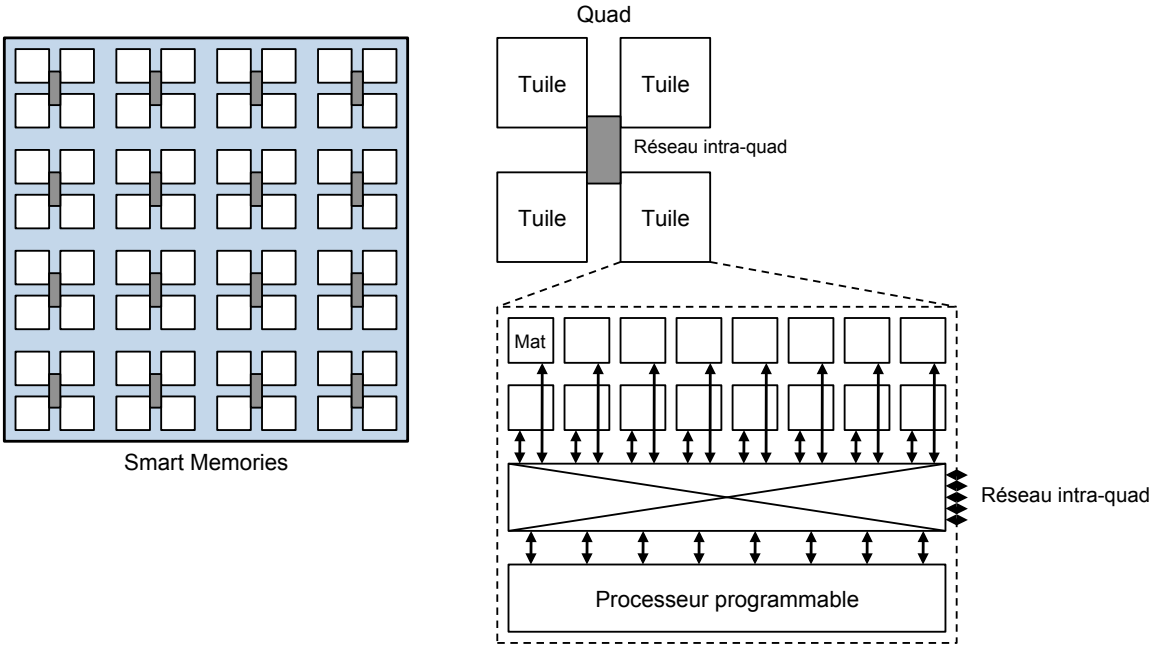


Fig. 1.25: *Architecture générale de Smart Memories.*

Dans une tuile, l'unité mémoire est partitionnée en 16 modules reconfigurables appelés *mats*. Un *mat* intègre une mémoire simple-port, de capacité 1024×64 bits, entourée de logique floue autorisant sa reconfiguration fonctionnelle (figure 1.26). Ainsi, sont disposés sur son chemin d'entrée et de sortie, respectivement, un registre pour le support d'opérations d'écriture conditionnelle et un comparateur qui permettent d'implémenter notamment des mémoires cache. Par ailleurs, une unité de génération d'adresses (*Stream registers and adder*, figure 1.26) admet des séquences très régulières à pas constants. Plus précisément, elle est composée de deux files de registres, stockant des valeurs de pointeurs et de pas, qui sont connectées à un additionneur-soustracteur pour la mise à jour des pointeurs. Dès lors, la multiplicité de registres disponibles autorise, avec un seul *mat*, la mise en œuvre conjointe de plusieurs primitives FIFO, qui accaparent chacune un pointeur d'écriture et un pointeur de lecture.

Au sein de la mémoire simple-port, chaque mot de 64 bits est associé à un champ de 5 bits qui est modifié grâce à un bloc de logique programmable. Par conséquent, en changeant la configuration de ce bloc, il est possible de changer dynamiquement la sémantique de ces cinq bits et d'y matérialiser, par exemple, un champ LRU[29] pour la mise en œuvre d'une politique de remplacement de blocs dans un cache particulière. En outre, les *mats* sont interconnectés par un réseau reconfigurable qui diffuse des informations de contrôle, tels des signaux *succès* (*hit*) et *échec* (*miss*) dans le cadre d'une solution cache, et qui permet aussi de construire des structures de capacités plus ou moins importantes.

Finalement, la reconfiguration fonctionnelle d'un *mat* est dynamique et réalisée grâce à des

[29]LRU : *Least Recently Used*

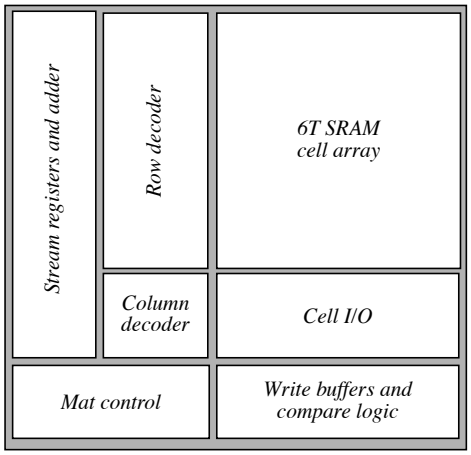


Fig. 1.26: *Organisation d'un mat (figure extraite de [43]).*

requêtes spécifiques émises par le cœur de processeur et décodées par une unité de contrôle interne au *mat* [45]. Pour une technologie 0.18 µm, un *mat* occupe une surface de 0.6 mm^2 et consomme une puissance égale à 125 mW pour une fréquence de fonctionnement de 1 GHz. Plus précisément, la mémoire occupe 68% de la surface d'un *mat* et correspond à 77% de sa consommation. À noter également que l'unité de génération d'adresses représente 25% de la surface de la mémoire simple-port et 19.5% de sa consommation d'énergie.

1.3 Synthèse

Aujourd'hui, les applications embarquées proposent de plus en plus de fonctionnalitées diverses et variées auxquelles s'adjoignent des contraintes de surface, de performance et de consommation. Pour répondre à ces défis, l'éclosion des architectures reconfigurables offre de nouveaux compromis dans le domaine de la conception électronique en matière de surface, d'efficacité énergétique et de flexibilité par rapport aux processeurs programmables et aux composants dédiés. Jusqu'à présent, l'étude de la mise en œuvre de la reconfiguration matérielle a permis d'optimiser, entre autre, la performance et la consommation des ressources opératives et de l'interconnexion. La mémoire, quant à elle, est bien souvent conçue selon une approche statique et hiérarchisée afin de lever le verrou du *Memory-Processor Gap*. Or, dans les systèmes sur silicium, les temps d'accès et la consommation des ressources de stockage dominent la performance et la dissipation énergétique du système. De surcroît, les applications de traitement de l'information induisent des motifs d'accès aux données multiples qui imposent une flexibilisation de la structure mémoire.

Dans ce contexte, plusieurs projets de recherche ont exploré l'application de la reconfigurabilité au niveau de la hiérarchie mémoire, de son interface avec les unités de calcul et de sa structure de génération d'adresses. Ainsi, la reconfiguration de la mémoire aboutit dans la plupart des cas à la possibilité de modifier sa fonctionnalité (par exemple le passage d'une mémoire à accès aléatoire à une structure FIFO voire une LUT) mais pas à l'opportunité d'adapter sa hiérarchie aux caractéristiques des traitements à réaliser. Concernant l'interface mémoire, les solutions proposées sont extrêmement flexibles et reposent sur des interconnexions globales de type *crossbar* ou multibus per-

mettant à toute ressource de calcul d'accéder à n'importe quel banc. Enfin, les unités de génération d'adresses sont développées dans le but de supporter des séquences exclusivement déterministes et ne présentant donc pas de dépendances de données et de contrôle. De manière générale, le contrôle des architectures reconfigurables ne convient pas aux applications dynamiques dont le comportement évolue en fonction des données traitées.

Dans le cadre de nos travaux de thèse, nous proposons donc une architecture reconfigurable dynamiquement qui cible les applications multimédia et de télécommunications mobiles. Celle-ci comprend notamment une hiérarchie mémoire flexible dont l'organisation est capable de s'adapter aux besoins des traitements à implémenter. De plus, cette structure de mémorisation intègre des AGU programmables supportant des séquences d'adresses régulières mais aussi irrégulières et exhibant des dépendances de données et de contrôle. Globalement, l'unité de contrôle de cette architecture a été élaborée de manière à pouvoir implémenter des applications dont le comportement change dépendamment des données manipulées. Le deuxième chapitre de ce mémoire décrit et justifie l'organisation de notre proposition d'architecture reconfigurable.

Chapitre 2

Description de MOREA

Ce deuxième chapitre présente notre proposition d'architecture reconfigurable dynamiquement et que nous avons nommée MOREA (acronyme de *Memory-Oriented Reconfigurable Embedded Architecture*). À cette fin, la première partie de ce chapitre expose le modèle d'application que nous considérons. Celui-ci justifie l'organisation générale de MOREA dont la description fonctionnelle est réalisée dans une deuxième partie. Puis, la structure des différents constituants de notre proposition architecturale est détaillée et justifiée en fonction des besoins des applications de traitement du signal et de l'image.

2.1 Modèle de spécification d'applications

Nous considérons des applications embarquées modélisées sous la forme de processus communiquants. L'espace mémoire de ces applications est distribué, c'est-à-dire que chaque processus dispose de sa propre mémoire de données privée. Les communications interprocessus consistent en des transferts de données impliquant plusieurs processus.

Un processus, quant à lui, est modélisé sous la forme d'un graphe de tâches (*threads*) qui accèdent concurremment à son espace mémoire (figure 2.1). Les communications inter-tâches consistent en des transferts de données impliquant plusieurs tâches et s'effectuent par l'intermédiaire de tampons (*buffers*) appartenant à l'espace mémoire du processus. Dès lors, on qualifie de tâches, toutes séquences d'opérations qui produisent ou consomment des données dans un ou plusieurs tampons.

Conjointement, on autorise qu'un tampon soit lu et écrit simultanément. Toutefois, il ne peut être produit que par une seule tâche. Cette restriction élimine tous risques d'écrasement des données produites par une tâche T_i par celles produites par une tâche T_j ainsi que la mise en œuvre d'une approche matérielle complexe pour la résolution de ce problème de validité des données dans le tampon. De plus, un tampon peut être consommé par plusieurs tâches dès lors que celles-ci ne s'exécutent pas de manière concurrente. En effet, la consommation des données dans un tampon par plusieurs tâches et selon des motifs distincts impose l'implémentation de mémoires multi-port très coûteuses en silicium. Cette dernière restriction entraîne par conséquent la transformation des divergences en ET dans un graphe de tâches. Par exemple, le graphe de la figure 2.2(a) illustre une telle divergence

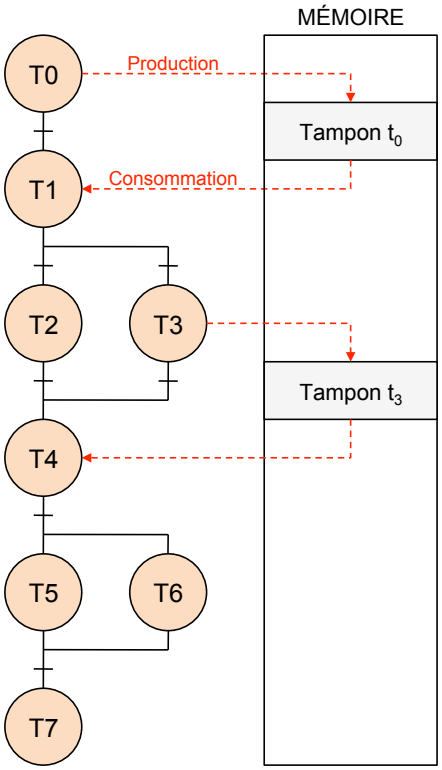


FIG. 2.1: *Exemple de processus modélisé par un graphe de tâches. Un processus est modélisé sous la forme d'un graphe de tâches qui accèdent concurremment à un espace mémoire privé. Les communications inter-tâches consistent en des transferts de données impliquant plusieurs tâches et s'effectuent par l'intermédiaire de tampons t_i appartenant à l'espace mémoire du processus.*

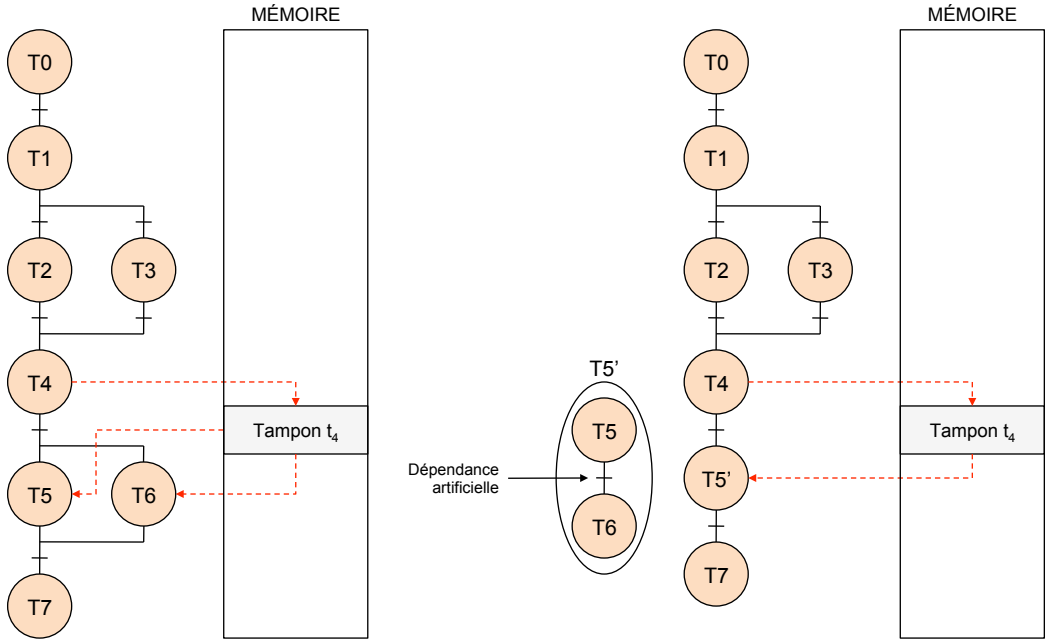
avec l'exécution parallèle des tâches T_5 et T_6. Dans ce cas de figure, une transformation se traduira par la fusion de ces tâches, via l'insertion d'une dépendance artificielle entre elles, et la création de la tâche $T_{5'}$ (figure 2.2(b)). De ce fait, le tampon produit par la tâche T_4 ne sera lu que par une seule tâche, en l'occurrence $T_{5'}$.

2.2 Organisation générale de MOREA

Les applications multimédia présentent plusieurs granularités de parallélisme (parallélisme au niveau processus, au niveau tâches, au niveau opérations et au niveau données). En conséquence, leur implémentation matérielle engendre des communications fortement localisées au niveau d'un processus ou d'une tâche. Aussi, l'architecture de MOREA est hiérarchisée et son interconnexion distribuée sur plusieurs niveaux afin de tenir compte de la localité des transferts de données et de réduire, par la même occasion, sa surface, ses délais de propagation et sa consommation d'énergie. Les paragraphes suivants décrivent fonctionnellement l'architecture de MOREA.

2.2.1 Architecture système de MOREA

L'architecture globale de MOREA est organisée en un pavage de tuiles (*tiles*) de traitement et de stockage (figure 2.3). Pour une application, chaque processus est exécuté par un cœur de



(a) Graphe initial (b) Graphe après transformation

Fig. 2.2: *Exemple de transformation d'une divergence en ET dans un graphe de tâches. (a) Graphe initial : les tâches T_5 et T_6 s'exécutent en parallèle et accèdent simultanément au tampon produit par la tâche T_4. (b) Graphe après transformation : les tâches T_5 et T_6 sont fusionnées au sein de la tâche $T_{5'}$. Désormais, une seule tâche, en l'occurrence $T_{5'}$, consomme les données du tampon produit par la tâche T_4.*

calcul. Celui-ci est une structure à gros grain, reconfigurable dynamiquement, qui comporte une interconnexion programmable supportant les échanges de données entre des unités opératives et de mémorisation. Les configurations des ressources d'une tuile relatives aux tâches du processus qu'elle exécute sont placées dans une mémoire de configuration intra-tuile. Par ailleurs, les tâches exécutées par une tuile étant ordonnancées dynamiquement, de nombreux transferts d'informations de la mémoire de configuration système (CM[1]), qui contient la totalité des configurations associées aux processus d'une application, vers les mémoires de configuration intra-tuile ont lieu pendant le traitement des différents processus.

Les applications embarquées induisent une multiplicité et une diversité de traitements auxquelles s'ajoutent des contraintes de performance importantes. Dès lors, une abondance de cœurs de calcul est nécessaire pour répondre aux besoins de ces applications. Cette complexité matérielle engendre un volume de communications conséquent qui ne peut être absorbé efficacement par les interconnexions traditionnelles à base de bus partagés. Aussi, dans MOREA, les communications inter-processus sont supportées par une infrastructure NoC [31]. Les transferts de données étant asynchrones, les différentes tuiles de MOREA sont connectées au NoC grâce à des interfaces (NI[3]) qui, entre autres, synchronisent les transactions au niveau système.

[1]CM : *Configuration Memory*
[3]NI : *NoC Interface*

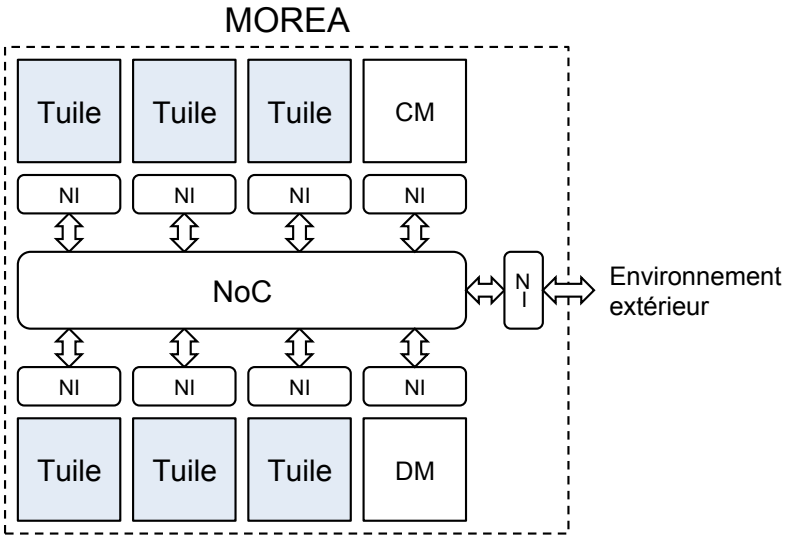


Fig. 2.3: *Architecture système de MOREA. Au plus haut niveau, MOREA est organisée en un pavage de tuiles de traitement et de stockage (CM et DM[2]). Chaque cœur de calcul est une structure à gros grain, reconfigurable dynamiquement, qui comporte une interconnexion programmable supportant les échanges de données entre des unités opératives et de mémorisation. Les différentes tuiles sont connectées à un réseau NoC, grâce à des interfaces (NI), qui assure les communications au niveau système.*

Finalement, des entrées/sorties spécifiques permettent à MOREA d'interagir avec un environnement extérieur. Par exemple, il peut s'agir d'un contrôleur externe responsable, entre autres, de l'assignation des processus de l'application sur les tuiles. Dans ce cas de figure, le contrôleur peut venir écrire dans la zone d'amorçage de la mémoire de configuration d'une tuile, permettant ainsi à cette dernière de récupérer de manière autonome dans le mémoire de configuration système les configurations correspondant au processus qui lui est assigné.

2.2.2 Architecture d'une tuile de MOREA

Une tuile de MOREA est structurée autour d'un réseau *crossbar* qui interconnecte quatre *clusters* exécutant les tâches d'un processus donné (figure 3.2). Ces *clusters* disposent de ressources de calcul et de stockage qui communiquent par l'entremise d'un motif flexible. Ces *clusters* accèdent de manière concurrente via le *crossbar* à une mémoire de tuile (TDM[4]) contenant les données partagées par les différentes tâches du processus. D'autre part, grâce à la programmabilité de l'interconnexion *crossbar*, l'exécution d'une tâche peut occuper un ou plusieurs *clusters*. Cette fonctionnalité permet dès lors de partager les données entre plusieurs tâches s'exécutant simultanément voire successivement et, en définitive, de limiter les transferts de données entre les ressources mémoire d'une tuile et donc de minimiser le temps d'exécution et la consommation de MOREA.

Un *cluster* est composé de quatre unités fonctionnelles (UF_i) qui sont interconnectées à quatre mémoires simple double-port (CDM_i[6]), c'est-à-dire admettant une opération de lecture et une opération d'écriture simultanément à chaque cycle, grâce à une structure multi-bus. Dans MOREA,

[4]TDM : *Tile Data Memory*
[6]CDM : *Cluster Data Memory*

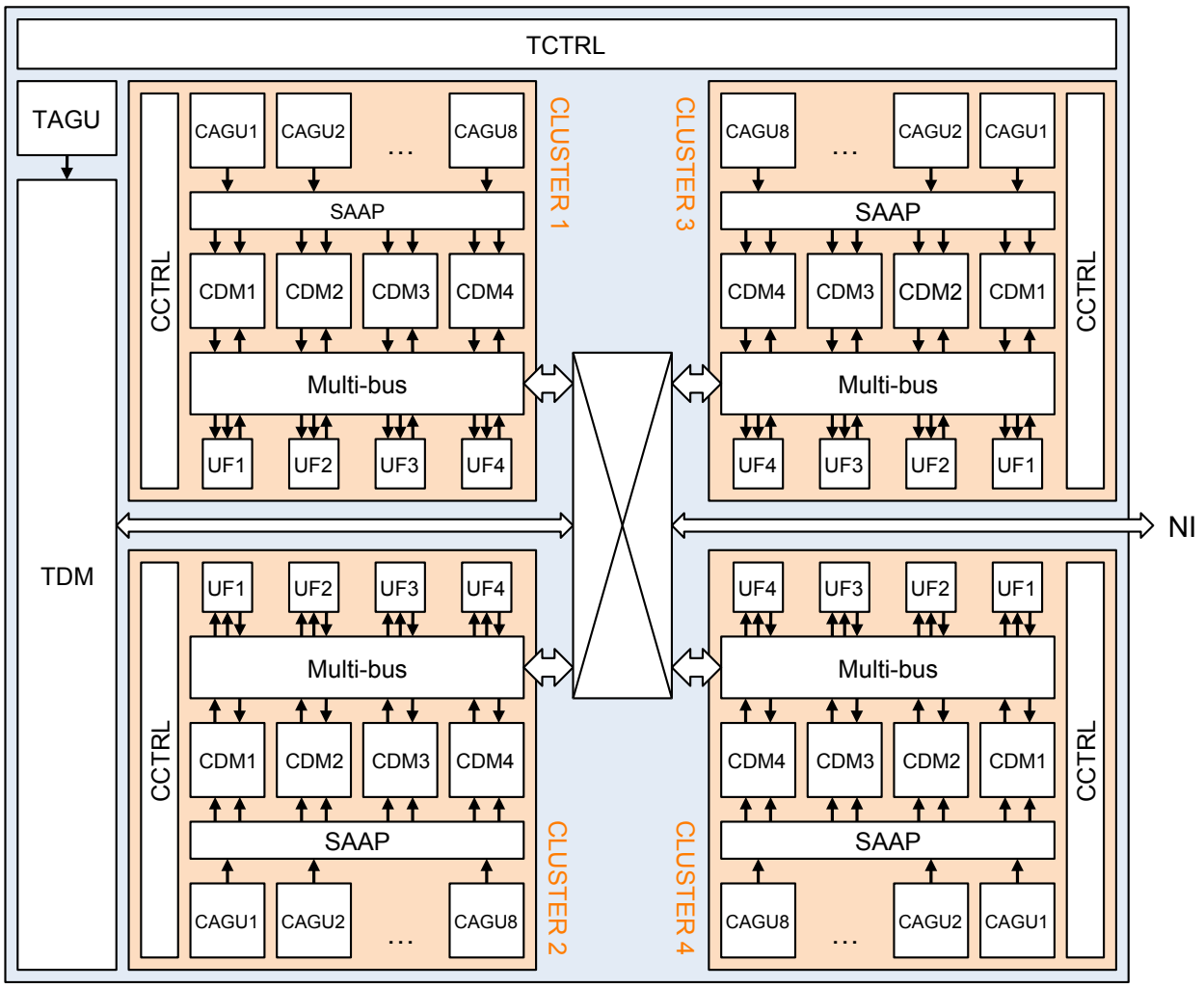


FIG. 2.4: *Architecture d'une tuile de MOREA. Elle est structurée autour d'un réseau crossbar qui interconnecte quatre clusters exécutant les tâches d'un processus donné. Ceux-ci accèdent de manière concurrente via le crossbar à une mémoire de tuile (TDM). Chaque cluster est composé de quatre unités fonctionnelles (UF_i) qui sont interconnectées à quatre mémoires simple double-port (CDM_i), c'est-à-dire admettant une opération de lecture et une opération d'écriture simultanément à chaque cycle, grâce à un motif flexible de type multi-bus. Dans une tuile, les accès aux données mémorisées sont commandées par des générateurs d'adresses programmables ($CAGU_i$ et TAGU) dont l'architecture leur permet de générer une grande variété de séquences d'adresses régulières et irrégulières. Finalement, la configuration des unités fonctionnelles et du réseau d'interconnexion ainsi que l'activation des générateurs d'adresses sont dévolues à une unité de contrôle hiérarchisée. Au plus haut niveau, un contrôleur de tuile (TCTRL) est responsable du placement des tâches, défini statiquement, et de leur ordonnancement dynamique. Il gère donc, via la programmation de l'interface NoC, la configuration des générateurs d'adresses et des contrôleurs de cluster ($CCTRL^5$). Ces derniers, quant à eux, sont chargés de la configuration des ressources intra-cluster.*

les accès aux données mémorisées sont commandées par des générateurs d'adresses programmables ($CAGU_i$[7] et TAGU[8]). Leur architecture leur permet de générer une grande variété de séquences d'adresses régulières et irrégulières. Enfin, dans une tuile, la configuration des unités fonctionnelles

[7]CAGU : *Cluster Address Generation Unit*
[8]TAGU : *Tile Address Generation Unit*

et du réseau d'interconnexion ainsi que l'activation des générateurs d'adresses sont dévolues à une unité de contrôle hiérarchisée. Au plus haut niveau, un contrôleur de tuile (TCTRL[9]) est responsable du placement des tâches, défini statiquement, et de leur ordonnancement dynamique. Il gère donc, via la programmation de l'interface NoC, la configuration des générateurs d'adresses et des contrôleurs de *cluster* (CCTRL[10]). Ces derniers, quant à eux, sont chargés de la configuration des ressources intra-*cluster*.

2.3 Architecture d'un *cluster*

Dans une tuile de MOREA, un *cluster* exécute les tâches d'un processus donné. Il est principalement constitué de quatre unités de calcul interconnectées à quatre mémoire de données par le biais d'une structure programmable de type multibus. L'architecture de l'unité de traitement, à savoir la quantité de ressources opératives disponibles et leur nombre d'entrées et de sorties, ainsi que les besoins exprimés par les motifs d'accès mémoire, c'est-à-dire notamment le nombre de lectures et d'écritures réalisées en parallèle, engendrés par les schémas de calcul observés dans les applications embarquées conditionnent le dimensionnement des ressources de mémorisation du *cluster*. De ce fait, nous présentons dans les paragraphes suivants, en premier lieu, la structure des unités fonctionnelles d'un *cluster* puis, celle de ses ressources de stockage.

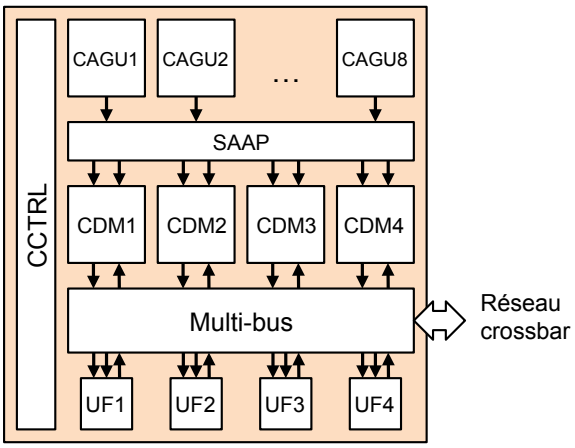


Fig. 2.5: *Architecture d'un cluster de MOREA. Un cluster est constitué de quatre unités fonctionnelles (UF_i) qui sont interconnectées à quatre mémoires simple double-port (CDM_i) grâce à une structure multi-bus. Les accès aux données mémorisées sont commandées par des générateurs d'adresses programmables ($CAGU_i$) dont l'architecture leur permet de générer une grande variété de séquences d'adresses régulières et irrégulières. La reconfiguration des unités opératives et du réseau d'interconnexions ainsi que l'activation des générateurs d'adresses sont effectuées par un contrôleur intra-cluster (CCTRL).*

2.3.1 Structure des ressources de calcul

L'architecture des unités fonctionnelles de MOREA provient de celle d'un DPR[11] du processeur reconfigurable DART [46, 12, 47] (figure 2.6). En effet, ce dernier cible les applications de téléphonie

[9]TCTRL : *Tile Controller*
[10]CCTRL : *Cluster Controller*
[11]DPR : *DataPath Reconfigurable*

mobile de troisième génération, impliquant donc des traitements multimédia, et qui répond par voie de fait à des contraintes similaires à celles induites par le domaine applicatif que nous considérons.

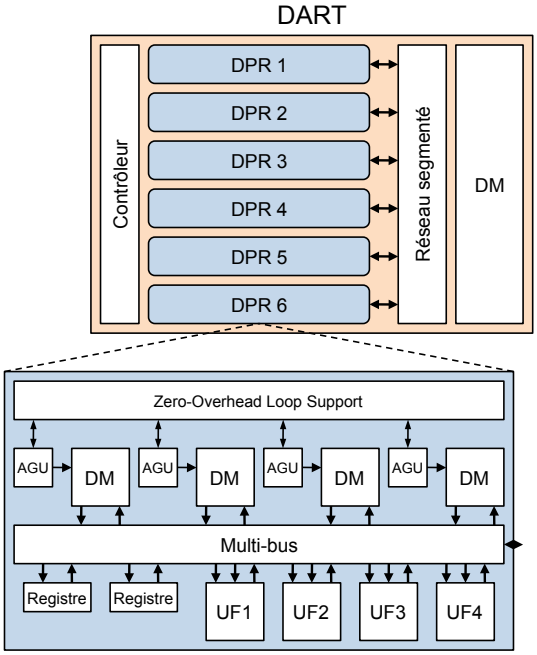


Fig. 2.6: *Architecture du processeur reconfigurable DART [12]. Elle est constituée de plusieurs cœurs de traitement (DPR) intégrant des ressources de calcul et de stockage. Plus précisément, un DPR dispose de quatre unités fonctionnelles (UF) à deux entrées et une sortie, dont deux multiplieurs-additionneurs (UF1 et UF3) et deux ALU (UF2 et UF4), qui réalisent des opérations sur des données 8, 16 ou* 32 bits. *Elles communiquent avec quatre mémoires (DM*[12]*) par l'entremise d'un réseau programmable de type multi-bus. Les accès aux données mémorisées sont déterminés par quatre générateurs d'adresses programmables (AGU*[13]*). D'autre part, une interconnexion segmentée permet de chaîner les opérateurs appartenant à des cœurs de traitement distincts, accroissant de ce fait leur puissance de calcul. Enfin, la reconfiguration dynamique des ressources de DART est effectuée par un contrôleur interne et n'excède pas quatre cycles d'horloge.*

Ainsi, chaque UF comprend deux entrées et une sortie. Dans un *cluster*, deux de ces UF, à savoir l'UF1 et l'UF3, implémentent, selon la configuration choisie, une multiplication de deux opérandes 16 bits avec un résultat obtenu en double précision, c'est-à-dire codé sur 32 bits, ou une addition sur deux données 32 bits. En outre, un décaleur à droite est placé à la sortie de l'opérateur afin, entre autre, de recadrer le résultat dans le cadre d'un traitement en virgule fixe.

L'UF2 et l'UF4, quant à elles, implémentent deux ALU qui réalisent donc des opérations arithmétiques (addition, soustraction, comparaisons et valeur absolue de la deuxième entrée) et des opérations logiques (AND, OR, XOR et NOP[14], c'est-à-dire que la sortie recopie la donnée de la première entrée) sur des données de largeur 32 bits. Par ailleurs, un décaleur bidirectionnel est disposé sur la première entrée et un décaleur à droite uniquement est inséré à la sortie de l'ALU. À cela s'ajoute également un registre en sortie de l'unité qui sert notamment d'accumulateur dans le cadre de la sélection du mode accumulation. Dans ce contexte, le chemin de données de l'UF est pourvu de huit

[14]NOP : *No OPeration*

bits de garde afin de prévenir les problèmes de débordement (*overflow*).

Finalement, le résultat d'une opération de comparaison (MIN ou MAX) est également généré sous une forme compressée grâce à un signal ou drapeau (*flag*) codé sur trois bits. Dans le domaine de l'embarqué, certaines applications présentent un caractère dynamique, c'est-à-dire que leur comportement varie en fonction des données traitées ou du résultat d'opérations réalisées sur ces données. Aussi, dans MOREA, les drapeaux provenant d'unités fonctionnelles de type 2 ou 4 sont connectés aux unités de contrôle de la tuile, à savoir les contrôleurs de configuration et les générateurs d'adresses, via des multiplexeurs et dans le but de matérialiser les dépendances de données exprimées par les traitements embarqués.

2.3.2 Architecture de l'unité de mémorisation

Dans un *cluster*, les mémoires de données CDM_i implémentent les tampons pour les communications inter-tâches. Elles approvisionnent en données les unités fonctionnelles et mémorisent les résultats des traitements réalisés. L'organisation de l'unité de stockage d'un *cluster* dépend donc de la structure de ses ressources de calcul mais également des motifs d'accès mémoire engendrés par les schémas de calcul observés dans les applications de traitement du signal et de l'image.

Ainsi, au sein de ce domaine applicatif, on dénombre une très grande variété de motifs de calcul dont nous pouvons citer quelques exemples :

– le motif MAC[15] (opération de convolution, produit matriciel) :

$$y = \sum_{i=1}^{N} a_i \times b_i \tag{2.1}$$

– le motif MAD[16] exhibé, par exemple, dans le cadre de la mise à jour des coefficients d'un filtre adaptatif :

$$y = a \times b + c \tag{2.2}$$

– le motif SAD[17] exploité dans les applications vidéo et, plus précisément, à des fins d'estimation du mouvement entre deux images successives ou pour la reconnaissance de formes particulières :

$$y = \sum_{i=1}^{N} |a_i - b_i| \tag{2.3}$$

– le motif ACS[18] mis en œuvre dans l'algorithme de Viterbi, c'est-à-dire dans le cadre d'une fonction de codage de canal au sein d'une chaîne d'émission en télécommunications :

$$y = (a_1 + a_2 > b_1 + b_2)\ ?\ a_1 + a_2\ :\ b_1 + b_2 \tag{2.4}$$

[15]MAC : *Multiply and Acumulate*
[16]MAD : *Multiply and Add*
[17]SAD : *Sum of Absolute Differences*
[18]ACS : *Add, Compare and Select*

– la multiplication complexe induite notamment par le calcul d'un papillon FFT :

$$y = (a_0 + j \cdot a_1) \times (b_0 + j \cdot b_1) = (a_0 \cdot b_0 - a_1 \cdot b_1) + j \cdot (a_0 \cdot b_1 + a_1 \cdot b_0) \tag{2.5}$$

Le tableau 2.1 présente les caractéristiques opératives et des accès mémoire des exemples de schémas de calcul listés précédemment.

	$\mathbf{Nb_{opérations}}$	$\mathbf{Nb_{lectures}}$	$\mathbf{Nb_{écritures}}$	$\frac{\mathbf{Nb_{lectures}}}{\mathbf{Nb_{opérations}}}$	$\frac{\mathbf{Nb_{écritures}}}{\mathbf{Nb_{opérations}}}$
MAC_N	$2 \cdot N$	$2 \cdot N$	1	1	0.1 (N = 4)
MAD	2	3	1	1.5	0.5
SAD_N	$3 \cdot N$	$2 \cdot N$	1	0.7	0.02 (N = 16)
ACS	3	4	1	1.3	0.3
MULT $\mathbb{C}$	6	4	2	0.7	0.3
Moyenne				1.04	0.24

Tab. 2.1: *Caractéristiques de quelques motifs de calcul extraits d'applications de traitement du signal et de l'image.*

D'après les données du tableau 2.1, les différents motifs étudiés génèrent en moyenne un accès mémoire en lecture et 0.24 accès en écriture par opération. Si nous supposons une implémentation spatialisée de ces traitements, c'est-à-dire qu'une opération est assignée à une unité fonctionnelle qui la réalise en un cycle machine, reflètant donc le pire cas d'exécution en terme de nombre d'accès mémoire parallèle à satisfaire, nous en déduisons qu'une UF génère un accès mémoire en lecture par cycle et 0.24 accès mémoire en écriture par cycle.

Pour répondre à ces besoins mémoire, outre les structures de stockage monolithiques et multiport dont le placement des données est transparent pour le programmeur mais qui se révèlent extrêmement gourmandes en surface, trois architectures multibanc peuvent être typiquement envisagées :

– les architectures fondées sur des mémoires simple-port, c'est-à-dire disposant d'un seul port de lecture et d'écriture,
– les architectures fondées sur des mémoires double-port, c'est-à-dire disposant de deux ports de lecture et d'écriture,
– les architectures fondées sur des mémoires simple double-port, c'est-à-dire disposant d'un port de lecture et d'un port d'écriture.

Les structures simple double-port sont rarement accessibles par le biais de bibliothèques fondeurs mais sont présentes dans les circuits FPGA sous la forme d'un mode de configuration particulier des mémoires flexibles proposées par ces composants. Aussi, dans le cadre de l'élaboration de MOREA, nous proposons une architecture de mémorisation simple double-port dont l'organisation est exposée via la figure 2.7.

Cette organisation comprend deux bancs simple-port dont la capacité respective est de 2^{m-1} mots pour une structure de capacité totale 2^m mots. La sortie d'un multiplexeur 2 : 1, dont les entrées

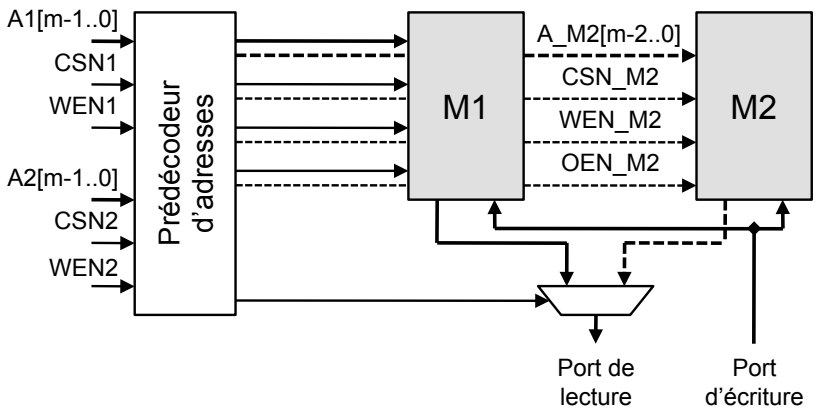
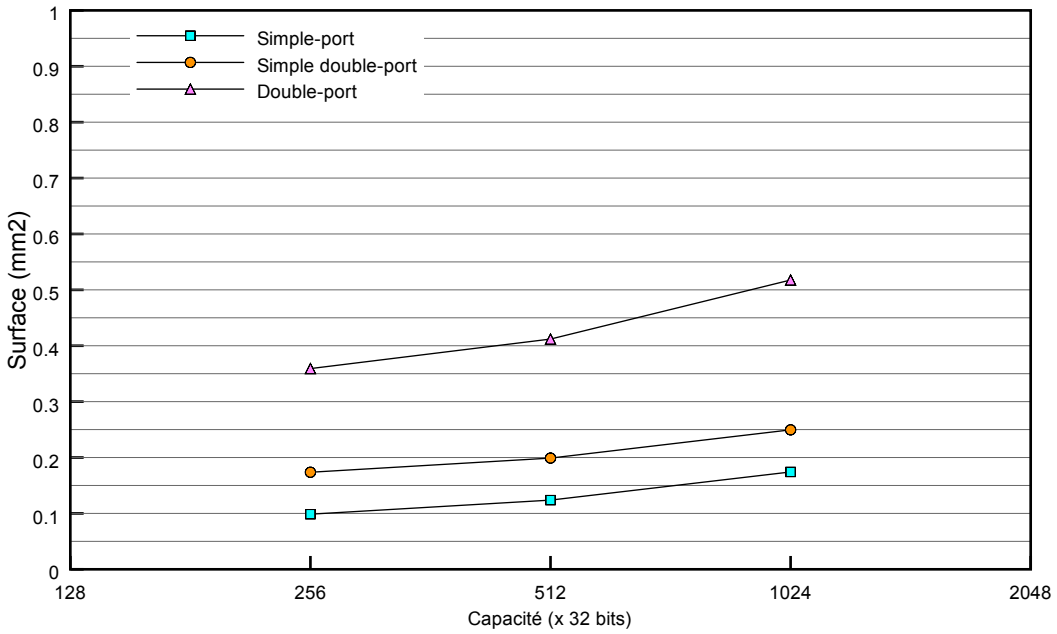


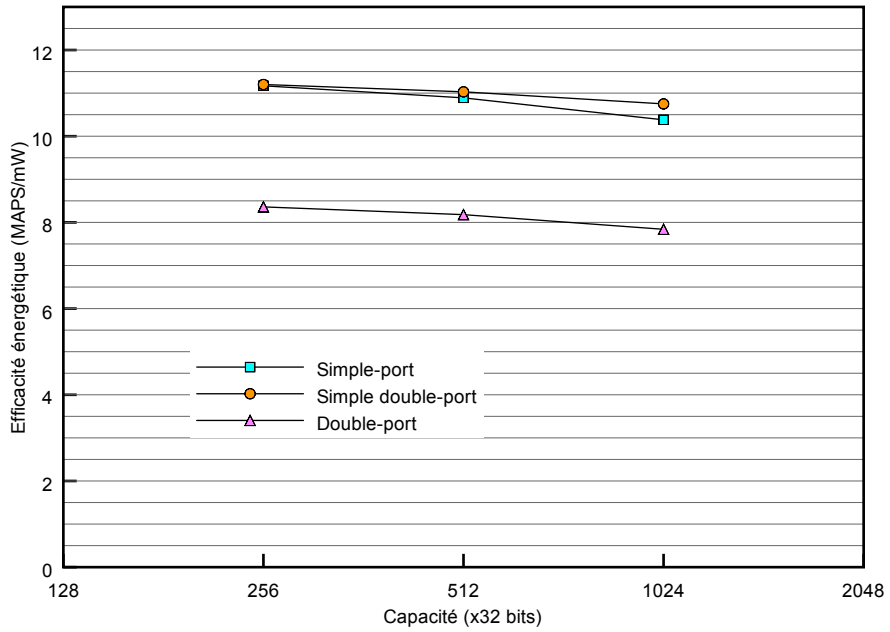
Fig. 2.7: *Architecture de la mémoire simple double-port de MOREA. Elle comprend deux bancs simple-port (M_i) dont la capacité respective est de 2^{m-1} mots pour une structure de capacité totale 2^m mots. La sortie d'un multiplexeur 2 : 1, dont les entrées sont connectées aux ports de lecture des deux mémoires simple-port, fournit la donnée lue. Les informations mémorisées dans cette architecture sont accédées par le biais de deux bus d'adresses symétriques (A_i), c'est-à-dire qui permettent de lire ou d'écrire dans l'un ou l'autre des deux bancs simple-port. D'autre part, pour chaque adresse émise, le bit de poids fort $A_i[m-1]$ détermine le banc simple-port accédé, c'est-à-dire si ce bit est égal à 0 (respectivement égal à 1) alors le premier banc (respectivement le deuxième banc) est accédé. Ce principe de fonctionnement est garanti par un pré-décodeur d'adresses.*

sont connectées aux ports de lecture des deux mémoires simple-port, fournit la donnée lue. Les informations mémorisées dans cette primitive simple double-port sont accédées par le biais notamment de deux bus d'adresses symétriques, c'est-à-dire qui permettent de lire ou d'écrire une donnée dans l'un ou l'autre des deux bancs simple-port. Cette propriété ajoute un caractère flexible à la primitive en permettant de l'utiliser en tant que structure simple-port, où un seul port d'adresses est exploité, ou en tant que structure simple double-port, où une adresse de lecture et une adresse d'écriture peuvent être décodées simultanément. D'autre part, pour chaque adresse émise, le bit de poids de fort détermine le banc simple port accédé, c'est-à-dire si ce bit est égal à 0 (respectivement égal à 1) alors le premier banc (respectivement le deuxième banc) est accédé. Ce principe de fonctionnement est garanti par un pré-décodeur d'adresses. Toutefois, celui-ci ne gère pas les conflits d'accès pouvant apparaître (par exemple l'émission de deux requêtes mémoire en lecture ou en écriture à la fois) et se contente de donner la priorité à la requête présentée sur le premier port d'adresses. Dans le cadre de la mise en œuvre d'un traitement sur MOREA, ces conflits d'accès sont résolus statiquement, c'est-à-dire à la compilation.

Les figures 2.8(a) et 2.8(b) montrent respectivement le coût en surface et en efficacité énergétique de structure de mémorisation simple-port, simple double-port et double-port. Dans le cas d'une architecture simple double-port, le prédécodeur d'adresses a été décrit en langage VHDL et synthétisé grâce à l'outil Synopsys Design Compiler pour une technologie ASIC 130 nm. À l'aide du script exposé en annexe B, cet outil a permis d'estimer la surface ainsi que la consommation d'énergie dans le pire cas de cette unité. Les informations relatives aux mémoires proprement dites ont été obtenues par le biais de compilateurs mémoire STMicroelectronics 130 nm.



(a)



(b)

Fig. 2.8: *Comparaison en surface (a) et en efficacité énergétique (b) de mémoires simple-port, simple double-port et double-port.*

Ainsi, en disposant dans un *cluster* quatre mémoires simple-port pour quatre unités fonctionnelles, la surface occupée par les ressources de stockage est minimale conjointement à l'obtention d'un niveau de performance et de consommation intéressants. Néanmoins, ce type d'architecture ne satisfait pas le nombre moyen de lecture et d'écriture induit par chaque unité fonctionnelle. En revanche, ce cas moyen ainsi que les pires cas exprimés via le tableau 2.1, à savoir 2 accès mémoire en

lecture et en écriture générés par opérateur dans le cadre du traitement d'un motif MAD et 1.6 accès générés par unité fonctionnelle dans le cadre de l'implémentation d'un motif ACS, sont satisfaits en utilisant un banc double-port par UF présente. Cependant, cette organisation occupe en moyenne une surface silicium trois fois plus importante que précédemment et affiche une efficacité énergétique en recul de 25%.

Une organisation à base de mémoires simple double-port satisfait la cas moyen déduit des données du tableau 2.1 mais pas les pires cas. De surcroît, la surface accaparée par cette solution est divisée par deux par rapport à celle occupée dans le cas de l'utilisation de bancs double-port. Cet avantage s'accompagne également d'une efficacité énergétique équivalente, voire sensiblement meilleure, que celle exhibée par une structure fondée sur des unités simple-port.

Finalement, pour des schémas de calcul nécessitant des mémoires double-port, deux solutions peuvent envisagées avec des structures simple double-port :

1. si plusieurs ressources mémoire sont encore disponibles, il est alors possible d'exploiter la flexibilité du réseau d'interconnexions de la tuile afin de solliciter ces ressources et de satisfaire les besoins en terme d'accès mémoire parallèle des traitements implémentés.
2. Sinon, pour le traitement concerné, il est indispensable de réduire le nombre d'opérations exécutées simultanément en les séquentialisant et donc en engendrant une baisse de la performance.

2.4 Description du réseau d'interconnexions

Dans une tuile de MOREA, les transferts de données sont supportés par un réseau hiérarchisé. Au plus haut niveau, les quatre *clusters*, la mémoire de données de la tuile et l'interface avec le réseau NoC sont interconnectés par une structure programmable de type *crossbar*. Dans un *cluster*, les communications entre les unités fonctionnelles et les ressources de stockage ainsi que les échanges de données avec les autres *clusters* sont mis en œuvre par un réseau multibus. Dans les paragraphes suivants, nous introduisons l'architecture de ces deux interconnexions.

2.4.1 Structure du réseau *crossbar*

Au sein d'une tuile, le réseau *crossbar* interconnecte les quatre *clusters* qui disposent chacun de M_{cluster} entrées et M_{cluster} sorties, la mémoire de données de la tuile simple port, qui dispose donc d'un port de lecture et d'un port d'écriture, et une interface avec le NoC qui comprend une entrée et une sortie vers le *crossbar*. Ce dernier est donc composé de $(2+4\cdot M_{\text{cluster}})$ entrées et $(2+4\cdot M_{\text{cluster}})$ sorties connectées à $(2+4\cdot M_{\text{cluster}})$ bus grâce à des *switches* (figure 2.9). Toutefois, afin de minimiser le nombre de *switches* présents, et donc la surface et les délais de propagation du réseau, tout en garantissant qu'une unité puisse toujours communiquer avec toutes les autres interconnectées par le réseau, deux restrictions sont imposées :

- une entrée du *crossbar* n'est connectée qu'à un seul bus donné,
- une sortie et une entrée appartenant à une même unité (par exemple un *cluster*) ne peuvent être reliées.

D'autre part, à un instant donné, une sortie du réseau n'est alimentée que par un et un seul bus. de ce fait, chaque entrée d'un *cluster* (respectivement de la mémoire de données ou de l'interface avec le NoC) est connectée, en réalité, à la sortie d'un multiplexeur $(2 + 3 \cdot M_{\text{cluster}}) : 1$ (respectivement $[1 + 4 \cdot M_{\text{cluster}}] : 1$).

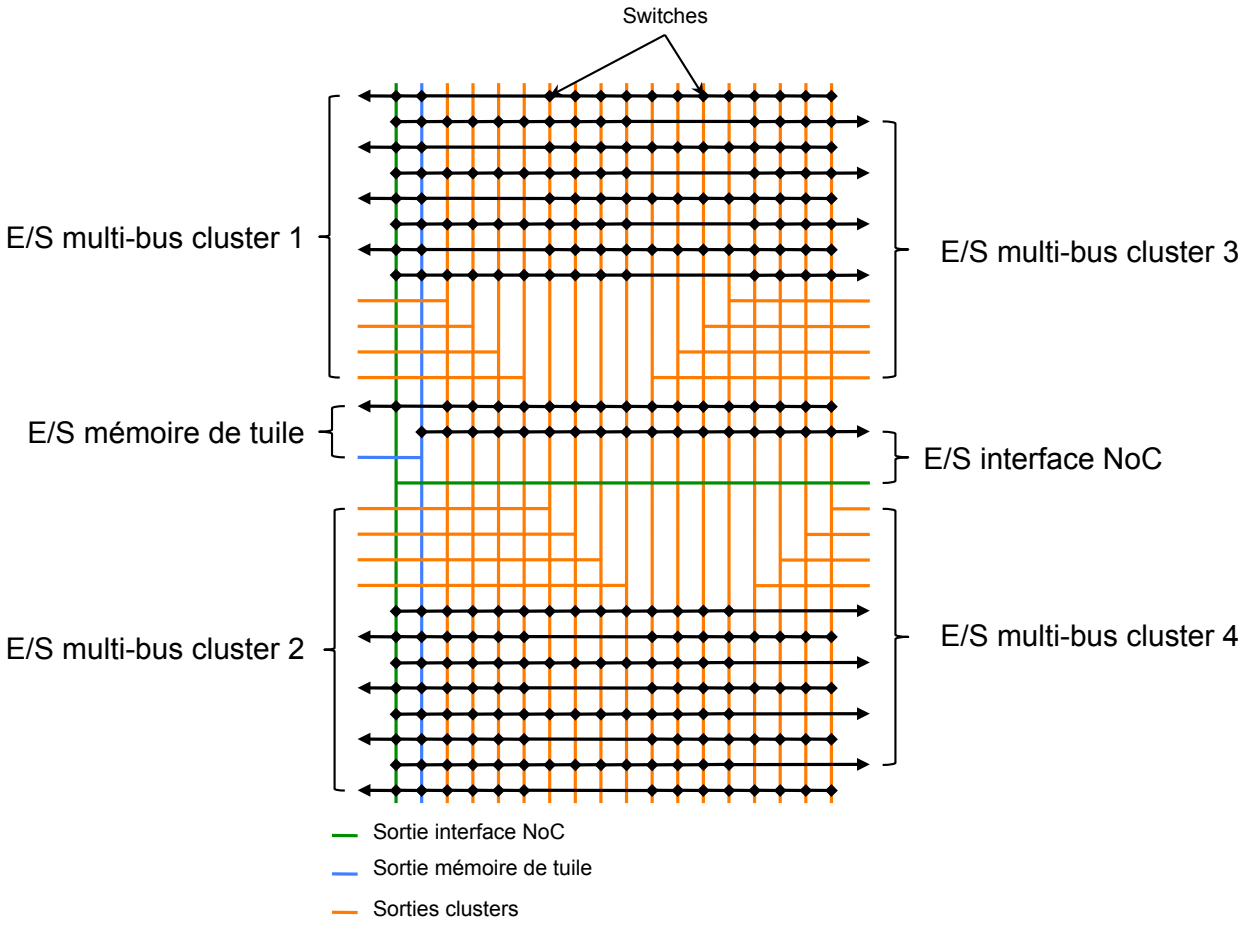


FIG. 2.9: *Architecture du réseau crossbar de MOREA pour $M_{cluster} = 4$. Cette structure interconnecte les quatre clusters qui disposent chacun de $M_{cluster}$ entrées et $M_{cluster}$ sorties, la mémoire de données de la tuile simple-port, qui dispose donc d'un seul port de lecture et d'un seul port d'écriture, et une interface avec le NoC qui comprend une entre et une sortie vers le crossbar. Ce dernier est donc composé de $(2 + 4 \cdot M_{cluster})$ entrées et $(2 + 4 \cdot M_{cluster})$ sorties connectées à $(2 + 4 \cdot M_{cluster})$ bus grâce à des switches.*

L'élaboration d'une structure de type *crossbar* assure une grande flexibilité en terme de schémas de communications entre tous les éléments qui y sont connectés, à savoir que toute unité peut communiquer avec n'importe quelle autre. Cette propriété s'accompagne, en outre, de temps de propagation uniformes qui simplifient le placement des tâches d'un processus sur une tuile. Cependant, cette structure est très gourmande en silicium et voit notamment sa surface croître de manière exponentielle en fonction du paramètre M_{cluster} et son temps de traversée augmenter linéairement en fonction de ce même paramètre. Par exemple, d'après les données du tableau 2.2, si le nombre d'entrées et de sorties par *cluster* double en passant de $M_{\text{cluster}} = 4$ à $M_{\text{cluster}} = 8$, alors la surface du réseau est multipliée par un facteur $\times 3.5$ tandis que sa performance diminue de 18%.

M_{cluster}	**Surface**	$t_{\textbf{propagation}}$
2	59443 μm^2	1.5 ns
4	140322 μm^2	1.7 ns
8	486018 μm^2	2 ns

Tab. 2.2: *Surfaces et performances du réseau crossbar de MOREA en fonction du nombre d'entrées et de sorties par cluster. Cette interconnexion a été décrite en VHDL générique, c'est à dire que le nombre d'entrées et de sorties par cluster ainsi que la largeur des bus de données sont deux paramètres de l'entité. Les données présentées dans ce tableau ont été estimées à partir de la synthèse sous Synopsys Design Compiler du réseau crossbar pour une technologie ASIC* 130 nm *et en considérant une largeur de bus égale à* 32 bits.

Dans une tuile de MOREA, le *crossbar* n'occupe que 14% de la surface dévolue à la tuile. Par ailleurs, cette surface dépend, entre autre, de la distance séparant les entrées et les sorties des quatre *clusters*. Aussi, dans le but de la minimiser, les quatre *clusters* sont disposés physiquement, c'est à dire au niveau *layout*, selon un point de symétrie, ce qui limite l'écart entre les entrées et les sorties des *clusters*.

2.4.2 Structure du réseau multi-bus

Dans un *cluster*, le réseau multi-bus remplit trois fonctions :

1. il interconnecte les unités fonctionnelles entre elles et de manière flexible afin de permettre l'implémentation de motifs de calcul variés,
2. il supporte les communications entre les unités fonctionnelles et les mémoires de données du *cluster*, et joue par conséquent le rôle d'interface mémoire,
3. il autorise les ressources du *cluster* à échanger des données avec celles des *clusters* environnants en présentant des entrées et des sorties communes avec celles du *crossbar*, ce qui permet donc notamment de répartir l'exécution d'une tâche sur plusieurs *clusters* en fonction de son degré de parallélisme d'opérations.

En conséquence, le réseau multibus est scindé en deux structures, à savoir un réseau admettant des communications entre les unités fonctionnelles et un autre réseau supportant les transactions entre les ressources de calcul et de stockage du *cluster* ainsi que les communications vers le réseau *crossbar* (figure 2.10).

Dans un *cluster*, l'interface mémoire interconnecte les quatre mémoires de données simple double-port qui disposent donc d'un port de lecture et d'un port d'écriture chacune, les quatre unités fonctionnelles comprenant chacune deux entrées et une sortie ainsi que le réseau *crossbar* via M_{cluster} entrées et M_{cluster} sorties. D'autre part, les applications de traitement du signal et de l'image présentent un comportement dynamique, c'est à dire que la nature des traitements réalisés évolue en fonction des données manipulées. Par conséquent, pour supporter cette caractéristique, le réseau d'interconnexion se doit de disposer d'une sortie spéciale permettant aux ressources de contrôle de la tuile, à savoir les contrôleurs de reconfiguration et les générateurs d'adresses, d'accéder aux informations traitées. Au niveau de l'interface mémoire, cette sortie est alimentée par les mémoires de

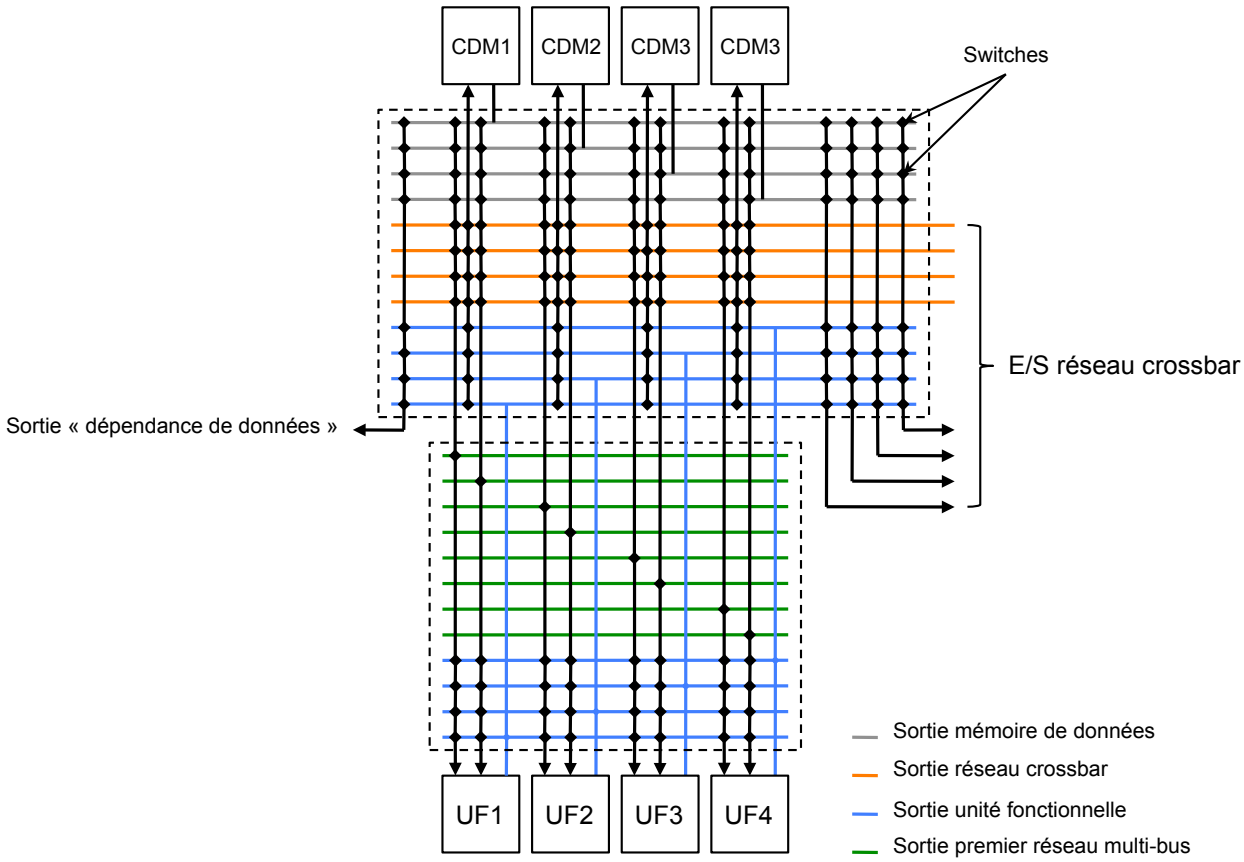


Fig. 2.10: *Architecture du réseau multi-bus de MOREA pour $M_{cluster} = 4$. Elle est scindée en deux structures, à savoir un premier réseau admettant des communications entre les unités fonctionnelles et un deuxième réseau supportant les transactions entre les ressources de calcul et de stockage du cluster ainsi que les communications vers le réseau crossbar. De plus, cette seconde interconnexion dispose d'une sortie spéciale (sortie dite « dépendance de données ») permettant aux ressources de contrôle de la tuile, à savoir les contrôleurs de reconfiguration et les générateurs d'adresses, d'accéder aux informations traitées par MOREA.*

données du *cluster* et les unités fonctionnelles.

Ainsi, cette interface mémoire, est constituée de $(8 + M_{\text{cluster}})$ entrées et de $(13 + M_{\text{cluster}})$ sorties connectées à $(8 + M_{\text{cluster}})$ bus grâce à des *switches*. De plus, cette interconnexion concède les mêmes restrictions que celles imposées au *crossbar* et auxquelles s'ajoute une dernière justifiant la dénomination multibus, à savoir que des unités « adjacentes » (par exemple deux mémoires de données ou deux unités fonctionnelles) ne peuvent communiquer entre elles.

De même qu'avec le motif *crossbar*, les entrées des unités fonctionnelles et des mémoires de données (respectivement du réseau *crossbar*) sont reliées à des multiplexeurs $(4 + M_{\text{cluster}}) : 1$ (respectivement $8 : 1$). En outre, tel qu'énoncé antérieurement, l'interface mémoire a deux rôles, à savoir la mise en œuvre des communications entre les ressources de calcul et de mémorisation et le support des transactions impliquant le *crossbar*. Ces deux fonctionnalités distinctes résultent en deux modes de reconfiguration partielle de l'interface. Par conséquent, chaque entrée d'une mémoire de données (respectivement d'une unité fonctionnelle) est approvisionnée par un bloc formé de trois

multiplexeurs :

1. un multiplexeur 4 : 1 qui sélectionne une donnée provenant d'une unité fonctionnelle (respectivement d'une mémoire de données),
2. un multiplexeur M_{cluster} : 1 qui sélectionne une donnée provenant du réseau *crossbar*,
3. un multiplexeur 2 : 1 qui sélectionne soit la donnée provenant de l'unité fonctionnelle (respectivement de la mémoire de données) ou soit de la structure *crossbar*.

Bien que présentant une flexibilité moindre par rapport à une structure *crossbar*, celle de l'interface mémoire reste néanmoins relativement importante. Globalement, ces deux types d'interconnexions présentent les mêmes avantages et les mêmes inconvénients, à la différence près, toutefois, que l'augmentation du nombre d'entrées et de sorties par *cluster* a un impact moins prononcé sur la surface et la performance du réseau multibus. Par exemple, d'après les données du tableau 2.3, le doublement de la valeur du paramètre M_{cluster}, en passant de $M_{\text{cluster}} = 4$ à $M_{\text{cluster}} = 8$, engendre un accroissement surfacique de 63% et une dégradation du temps de propagation de 7%. Globalement, la surface combinée des quatre réseaux multibus de la tuile ne représente que 4% de celle-ci.

M_{cluster}	**Surface**	$t_{\textbf{propagation}}$
2	56730 μm^2	1.4 ns
4	74285 μm^2	1.4 ns
8	121099 μm^2	1.5 ns

Tab. 2.3: *Surfaces et performances de l'interface mémoire en fonction du nombre d'entrées et de sorties par cluster.*

Le deuxième réseau multibus, quant à lui, interconnecte les quatre unités fonctionnelles entre elles. Il est constitué de huit entrées et de quatre sorties avec l'interface mémoire et de huit sorties et quatre entrées avec les unités fonctionnelles, qui sont connectées à 12 bus grâce à des *switches*. Cette interconnexion réalise deux fonctions, à savoir permettre les transferts de données entre les unités de calcul et acheminer les communications impliquant l'interface mémoire. Dès lors, les entrées des unités fonctionnelles sont issues d'une logique de multiplexage intégrant les deux éléments suivants :

1. un multiplexeur 4 : 1 qui sélectionne une donnée provenant d'une unité fonctionnelle,
2. un multiplexeur 2 : 1 qui sélectionne soit la donnée provenant d'une unité fonctionnelle ou soit émanant de la sortie correspondante de l'interface mémoire.

Finalement, le réseau de la tuile interconnecte les différentes ressources mémoire et ce quelle que soit le niveau de hiérarchie auquel elles appartiennent. Ainsi, la programmabilité de cette interconnexion permet d'adapter la hiérarchie mémoire en fonction des besoins de l'application implémentée. Dans le cadre du troisième chapitre et de la validation de MOREA, nous démontrerons la flexibilité de la hiérarchie mémoire d'une tuile et de son intérêt pour la mise en œuvre d'applications de traitement du signal et de l'image.

2.5 Organisation de l'unité de génération d'adresses

Dans une tuile, les accès aux données mémorisées sont commandés par des générateurs d'adresses, à savoir un générateur pour la mémoire de données simple port de la tuile et deux AGU pour chaque mémoire simple double port des *clusters*. D'autre part, les applications de traitement de l'information présentent une grande variété de séquences d'adresses qui requièrent donc l'élaboration d'unités de génération d'adresses flexibles. Aussi, dans MOREA, les AGU sont fondées sur des solutions programmables émanant du concept RISC [48]. Globalement, elles sont donc constituées d'une unité de contrôle organisée autour d'une mémoire d'instructions dont la lecture puis le décodage permettent notamment de configurer une unité de calcul qui détermine les valeurs successives de la séquence d'adresses. Les paragraphes suivants détaillent la structure de ces deux parties.

2.5.1 Architecture de l'unité de calcul de l'AGU

Dans les applications multimédia, les données sont généralement organisées sous la forme de tableaux mono ou multidimensionnels qui sont accédés par l'entremise de nids de boucles imbriquées. L'adresse d'une case du tableau est alors définie par une fonction des indices de boucles. Par conséquent, l'unité de calcul d'un générateur d'adresses incorpore une file de huit registres de largeur N_{adresse} bits, avec N_{adresse} la dimension du bus d'adresses de la mémoire accédée (figure 2.11). Cette file permet notamment de stocker la valeur courante de quatre indices de boucles plus le résultat d'une fonction celles-ci. En outre, elle est équipée d'une entrée permettant de recevoir une valeur immédiate, c'est à dire directement issue du décodage d'une instruction, la sortie d'une ALU et également une donnée provenant du réseau d'interconnexions de la tuile. Cette dernière particularité autorise la mise en œuvre de séquences d'adresses dépendantes des données de l'application. Finalement, les données stockées dans cette file sont lues grâce à deux sorties qui alimentent, entre autre, une unité arithmétique et logique.

Cette ALU réalise à chaque période d'horloge une opération sur une ou deux opérandes de taille N_{adresse} bits. Plus précisément, elle supporte l'addition et la soustraction de deux valeurs positives ainsi qu'une opération ET logique autorisant l'implémentation d'opérations de masquage d'adresses. De surcroît, elle est capable d'effectuer des décalages à droite comme à gauche et d'amplitude 1, 2 ou 4 bits sur sa première entrée. Notons que les générateurs d'adresses de MOREA ne disposent pas de bloc spécifique proposant l'opération *bit-reverse*, fréquemment rencontrée dans les applications de télécommunications mobiles via la transformée de Fourier rapide ou FFT. Néanmoins, la disponibilité des opérations ET logique et de décalages permettent, tel que démontré par le code exposé en annexe A, d'exécuter une version dégradée de cette fonction.

Les séquences d'adresses exhibées par les applications de traitement du signal et de l'image peuvent également présenter des dépendances de contrôle, c'est à dire que le choix d'une adresse particulière parmi plusieurs valeurs possibles est conditionné par le résultat d'une opération antérieure. Pour supporter ces schémas, l'ALU génère en outre deux signaux ou drapeaux (*flags*) à savoir :

– la retenue (*Carry*) engendrée par l'opération réalisée et qui indique, suite à une soustraction

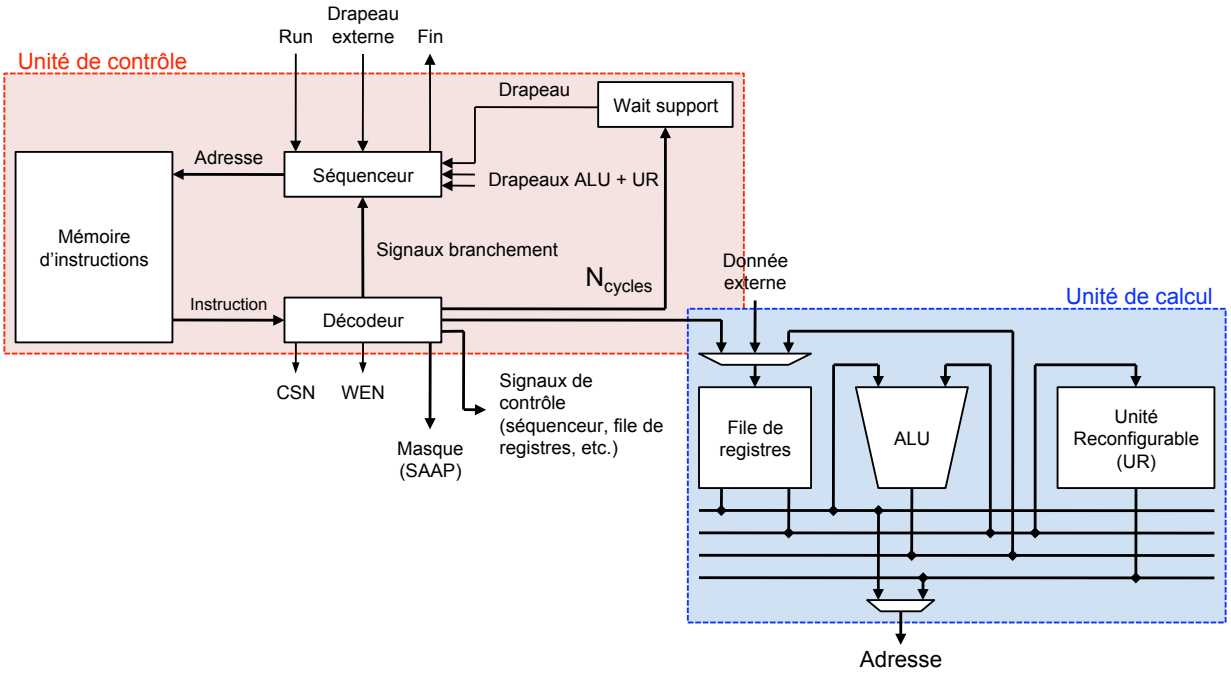


FIG. 2.11: *Architecture du générateur d'adresses de MOREA.*

notamment, si le résultat est positif (C = 0) ou négatif (C = 1)

– un indicateur précisant si la valeur obtenue est nulle ou non.

Dès lors, ces deux drapeaux sont analysés par l'unité de contrôle de l'AGU au moment de la validation ou pas d'un branchement conditionnel.

De manière générale, dans les traitements multimédia, de nombreux motifs d'adresses sont affines, c'est à dire qu'ils sont modélisés par une combinaison affine des indices de boucles :

$$\mathrm{AE} = C_0 + \sum_{j=1}^{N} C_j \cdot i_j \tag{2.6}$$

où $\{C_k\}$ sont des constantes, dont C_0 représente la première adresse de la séquence, et $\{i_k\}$ sont les indices de boucles. Ainsi, ces séquences sont extrêmement régulières car présentant des calculs très répétitifs. Leur mise en œuvre matérielle peut donc être optimisée relativement aisément grâce à l'introduction d'une unité opérative dédiée. Dans MOREA, à l'image du processeur Chimaera [49], chaque générateur d'adresses intègre un bloc reconfigurable destiné au traitement des séquences d'adresses affines. La brique élémentaire de ce module est un *stepper* dont l'architecture est inspirée d'un élément de même nom et appartenant à l'unité de génération d'adresses de l'architecture MoM[19] [50] (figure 2.12). Celle-ci est destinée aux domaines du traitement de l'image et de la vidéo qui présentent des motifs d'accès complexes de types zig-zag ou spirale par exemple.

Concrètement, un *stepper* génère une séquence d'adresses affine fonction d'un seul indice de boucle. Soit $\mathrm{AE}(i_1) = C_0 + C_1 \cdot i_1$ et l'indice i_1 dont le comportement est représenté par le triplet

[19]MoM : *Map-oriented Machine*

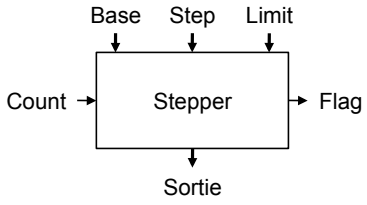


FIG. 2.12: *Vue générale d'un stepper.*

$(\alpha_1, \delta_1, \omega_1)$, où α_1 est la valeur initiale de l'indice i_1, δ_1 son pas d'incrémentation ou de décrémentation et ω_1 sa valeur finale. Dans ce cas, les entrées du *stepper* sont définies telles que :

$$\begin{cases} \text{Base} = \text{AE}(\alpha_1) = C_0 + C_1 \cdot \alpha_1 \\ \text{Step} = \text{AE}(i_1 + \delta_1) - \text{AE}(i_1) = C_1 \cdot \delta_1 \\ \text{Limit} = \text{AE}(\omega_1) = C_0 + C_1 \cdot \omega_1 \end{cases} \tag{2.7}$$

Dès lors, le *stepper* démarre la génération de la séquence d'adresses, au rythme d'une valeur par cycle d'horloge, après affirmation de l'entrée *Count* (Count = 1). Au moment du calcul de la dernière adresse ou, de manière équivalente, au moment de la dernière itération de la boucle d'indice i_1, un drapeau *Flag* est levé (Flag = 1).

À présent, si l'on considère la fonction $\text{AE}(i_1, i_2) = C_0 + C_1 \cdot i_1 + C_2 \cdot i_2$, les entrées du *stepper* sont redéfinies de la manière suivante :

$$\begin{cases} \text{Base} = \text{AE}(\alpha_1, i_2) = \text{AE}_{\alpha_1}(i_2) = C_0 + C_1 \cdot \alpha_1 + C_2 \cdot i_2 \\ \text{Step} = \text{AE}(i_1 + \delta_1, i_2) - \text{AE}(i_1, i_2) = C_1 \cdot \delta_1 \\ \text{Limit} = \text{AE}(\omega_1, i_2) = \text{AE}_{\omega_1}(i_2) = C_0 + C_1 \cdot \omega_1 + C_2 \cdot i_2 \end{cases} \tag{2.8}$$

Dans ce deuxième cas de figure, les entrées *Base* et *Limit* ne sont plus constantes et proviennent de deux séquences d'adresses affines qui dépendent de l'indice i_2. La structure permettant d'implémenter la fonction $\text{AE}(i_1, i_2)$ est donc constituée d'un premier *stepper* Stepper_1 dont les entrées Base_1 et Limit_1 sont connectées respectivement à la sortie de deux autres *steppers* Stepper_2 et Stepper_3 (figure 2.13). Les entrées globales de cette nouvelle structure sont alors déterminées de la façon suivante :

$$\begin{cases} \text{Step}_1 = \text{AE}(i_1 + \delta_1, i_2) - \text{AE}(i_1, i_2) = C_1 \cdot \delta_1 \\ \text{Base}_2 = \text{AE}_{\alpha_1}(\alpha_2) = C_0 + C_1 \cdot \alpha_1 + C_2 \cdot \alpha_2 \\ \text{Step}_2 = \text{AE}_{\alpha_1}(i_2 + \delta_2) - \text{AE}_{\alpha_1}(i_2) = C_2 \cdot \delta_2 \\ \text{Limit}_2 = \text{AE}_{\alpha_1}(\omega_2) = C_0 + C_1 \cdot \alpha_1 + C_2 \cdot \omega_2 \\ \text{Base}_3 = \text{AE}_{\omega_1}(\alpha_2) = C_0 + C_1 \cdot \omega_1 + C_2 \cdot \alpha_2 \\ \text{Step}_3 = \text{AE}_{\omega_1}(i_2 + \delta_2) - \text{AE}_{\omega_1}(i_2) = C_2 \cdot \delta_2 \\ \text{Limit}_3 = \text{AE}_{\omega_1}(\omega_2) = C_0 + C_1 \cdot \omega_1 + C_2 \cdot \omega_2 \end{cases} \tag{2.9}$$

D'autre part, dans le cas de l'exécution de deux boucles imbriquées, l'indice de la boucle externe i_2 est mis à jour après la dernière itération de la boucle interne d'indice i_1. Au niveau matériel, ce comportement se traduit par l'interconnexion de la sortie Flag_1 du *stepper* Stepper_1 à l'entrée *Count*

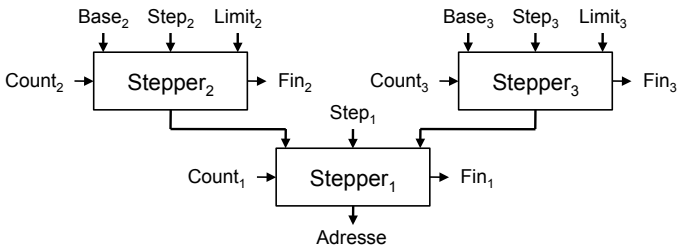


FIG. 2.13: *Exemple de structure à base de steppers pour la génération de séquences d'adresses affines fonctions de deux indices de boucles.*

respective des deux autres *steppers*. Finalement, la génération de la séquence d'adresses s'achève suite à la dernière itération de la boucle d'indice i_1 lors de la dernière itération de la boucle d'indice i_2. Le signal *Flag* global résulte donc du ET logique des trois sorties du même nom appartenant aux trois *steppers*. Par conséquent, la règle de conception suivante peut être posée : « Une séquence d'adresses affine fonction de N indices de boucles est mise en œuvre par un arbre binaire de *steppers* de hauteur N où les entrées *Base* et *Limit* (respectivement la sortie *Flag*) d'un nœud de profondeur n sont connectées aux sorties (respectivement aux entrées *Count*) de ses deux nœuds fils de profondeur $n+1$. Le signal *Flag* global résulte du ET logique des sorties du même nom des $2^N - 1$ *steppers* de l'arbre. »

Dans MOREA, le nombre de *steppers* intégrés par générateur d'adresses est restreint à trois et permet donc de générer toutes séquences d'adresses fonction de deux indices de boucles au plus (figure 2.14). Pour des séquences dépendant de plus de deux indices, les indices supplémentaires sont gérés de manière logicielle. En outre, l'insertion notamment de multiplexeurs commandés par un bit de configuration permet de flexibiliser l'architecture en offrant la possibilité à l'utilisateur d'exploiter un ou trois *steppers*.

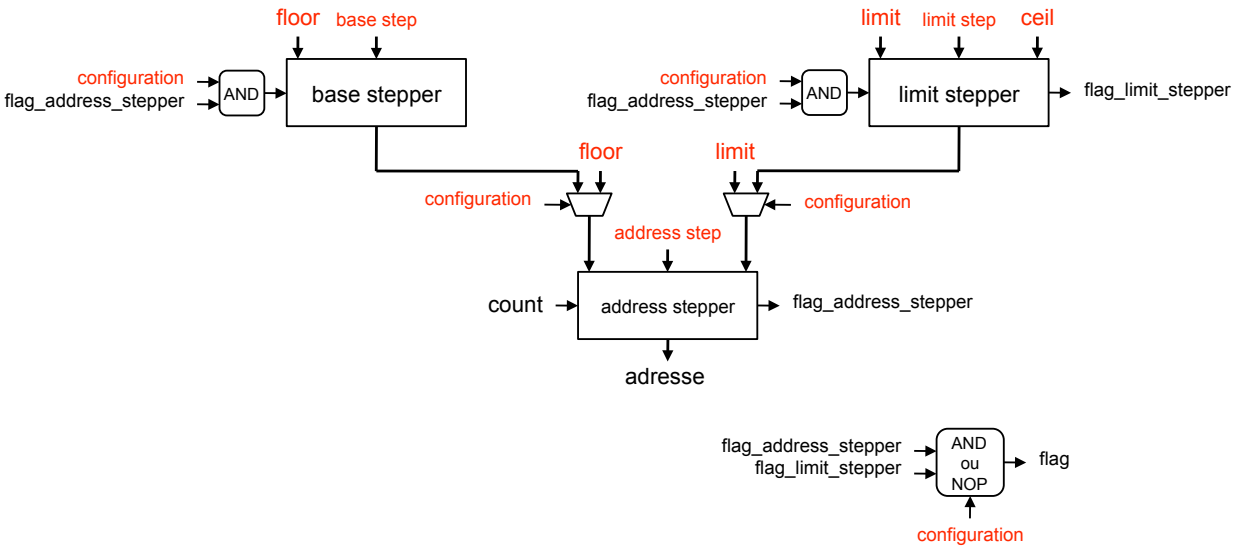


FIG. 2.14: *Architecture de l'unité reconfigurable du générateur d'adresses de MOREA.*

2.5.2 Architecture de l'unité de contrôle de l'AGU

Dans MOREA, une séquence d'adresses est représentée par une suite de directives compréhensibles pour un générateur d'adresses et stockées dans une mémoire d'instructions. Les instructions suivent un chemin pipeliné et constitué de quatre étages. Plus précisément, à chaque cycle machine, une instruction dont l'adresse est positionnée dans le compteur programme d'un séquenceur est lue. Dès lors, celle-ci est interprétée par une unité de décodage au cycle d'après. Au cycle suivant, les signaux générés par le décodeur d'instruction sont propagés vers l'unité de calcul du générateur et déterminent, par exemple, l'opération réalisée par l'ALU ainsi que les registres contenant les opérandes à traiter et celui mémorisant le résultat obtenu. Finalement, au cours d'une quatrième étape, la sortie de l'unité arithmétique et logique est placée dans un registre destination.

Le jeu d'instructions d'un AGU comprend 16 instructions (tableau 2.4). Celles-ci permettent ainsi de charger un registre de la file avec soit une donnée immédiate, c'est à dire directement extraite du décodage d'une instruction LOAD, ou soit une donnée externe à l'AGU, c'est à dire provenant du chemin de données de la tuile, grâce à la directive GET. D'autre part, les instructions ADD, SUB, AND et ASH configurent l'ALU pour effectuer respectivement soit une addition, soit une soustraction, soit une opération ET logique ou bien une opération de décalage. Dans ce dernier cas, l'opération est appliquée à la valeur du deuxième registre mentionné dans l'instruction et le résultat est placé dans un registre destination qui peut être différent du registre source. De plus, le dernier argument précise le sens du décalage, c'est à dire à droite (-) ou à gauche (+) et son amplitude qui peut être de 1, 2 ou 4 bits.

Code opératoire	**Instruction**	**Syntaxe**	**Description**
1	LOAD	RAn, < opérande >	Donnée immédiate → RAn
2	GET	RAn	Donnée externe → RAn
3	CONF	RXn, RAm	RAm → RXn
4	OUT	{W, R} RAn	RAn → @
5	ADD	RAx, RAy, RAz	RAy + RAz → RAx
6	SUB	RAx, RAy, RAz	RAy − RAz → RAx
7	AND	RAx, RAy, RAz	RAy · RAz → RAx
8	ASH	RAx, RAy, $\pm d$	*Arithmetic Shift*
9	ROP	{W, R}, (< N_{cycles} >)	*Reconfigurable Operation*
10	BNZ	< étiquette >	*Branch if Not Zero*
11	BCS	< étiquette >	*Branch if Carry Set*
12	BXF	< étiquette >	*Branch if External Flag Set*
13	BRA	< étiquette >	*Branch Always*
14	WAIT	< N_{cycles} >	*Wait State*
15	END		Programme terminé
0	NOP		*No Operation*

Tab. 2.4: *Description du jeu d'instructions de générateur d'adresses de MOREA.*

Les instructions CONF et ROP permettent d'exploiter l'unité reconfigurable du générateur d'adresses. Concrètement, la première instruction détermine le contenu d'un registre de configuration (ou auxiliaire) à partir de la valeur d'un registre appartenant à la file (tableau 2.5). Puis, la deuxième instruction précise le type d'opérations mémoire à réaliser, c'est à dire soit une suite

de lectures (R[20]) ou une suite d'écritures (W[21]), et lance le calcul de la séquence d'adresses. Cette phase de traitement peut soit s'achever dès l'affirmation du drapeau de l'unité reconfigurable ou bien, si le nombre de valeurs à générer est connu à l'avance ou si la séquence ne dépend pas d'une donnée applicative, peut se terminer au bout d'une période de temps définie statiquement. De surcroît, lors du calcul des adresses par l'unité reconfigurable, le pipeline du générateur d'adresses est bloqué et aucune instruction supplémentaire n'est lue ni décodée. Par conséquent, la consommation dynamique de l'AGU est dominée par celle de son module reconfigurable et non plus par celle de sa mémoire d'instructions. Cette caractéristique réduit de manière substantielle l'énergie dissipée par l'architecture dans le cadre de la mise en œuvre de séquences d'adresses affines comme nous le démontrerons au chapitre suivant.

Registre	**Nom du signal**
RX0	Floor
RX1	Address Step
RX2	Limit
RX3	Base Step
RX4	Limit Step
RX5	Ceil
RX6	Configuration

Tab. 2.5: *Liste des registres auxiliaires du générateur d'adresses de MOREA.*

Puisque certains schémas d'adresses présentent des dépendances de contrôle, le jeu d'instructions des AGU inclut des directives de branchement conditionnel (instructions BNZ, BCS et BXF) et inconditionnel (instruction BRA). L'instruction BXF permet notamment de tester la valeur d'un drapeau généré par une unité fonctionnelle de la tuile et de modifier en conséquence le séquencement des instructions. D'autre part, telle qu'illustrée par la figure 2.15, la disponibilité d'une directive de branchement inconditionnel est obligatoire pour implémenter des structures de programmation de type *if-then-else*.

Finalement, pour permettre la synchronisation des diverses unités programmables intégrées dans une tuile, celles-ci proposent dans leur jeu d'instructions des commandes WAIT et NOP permettant d'insérer dans l'exécution du programme des cycles d'attente. À noter que l'instruction WAIT fige le pipeline de l'unité pendant un nombre fini de périodes d'horloge, éliminant ainsi toute consommation relevant de la lecture et du décodage d'instruction. Pour sa part, l'instruction END stoppe le fonctionnement de l'unité programmable et lève un drapeau qui peut être traité ensuite par un contrôleur dans le cadre de la synchronisation des différentes opérations imposées par l'application. Pour un générateur d'adresses, la taille des instructions est à priori déterminée par celle de la directive LOAD et est égale à $N_{\text{adresse}} + 7$ bits.

[20] R : *Read*
[21] W : *Write*

```
1  if(a < b){
2     read(a);
3  }
4  else{
5     read(b);
6  }
```

(a)

```
1         GET RA0 ; RA0 = a
2         GET RA1 ; RA1 = b
3         SUB RA2, RA0, RA1 ; a - b
4         BCS etq1 ; a < b ?
5         OUT R, RA1 ; read(b)
6         BRA etq2
7   etq1: OUT R, RA0 ; read(a)
8   etq2: < suite >
```

(b)

Fig. 2.15: *Traduction en langage d'assemblage (b) d'un code C (a) comportant une structure conditionnelle if-then-else.*

2.5.3 Structure de l'interface SAAP

Les applications multimédia exhibent plusieurs types de parallélisme dont celui de données. Ce dernier implique des traitements de type SIMD où une même opération est appliquée à plusieurs données en parallèle. Dans ce cas de figure, dans un *cluster*, deux mémoires de données ou plus sont accédées suivant un même motif d'adresses. Dès lors, l'allocation fixe d'un générateur d'adresses par port d'adresses disponible impose de programmer et d'activer plusieurs AGU engendrant de ce fait un coût en temps d'exécution et en consommation énergétique important. Par conséquent, dans un souci d'efficacité énergétique, les générateurs d'adresses d'un *cluster* sont connectés aux $N_{\text{mémoires}}$ mémoires de données par le biais d'une interconnexion flexible (figure 2.16). Celle-ci, dénommée SAAP, achemine dynamiquement les adresses émises par les AGU vers un ou plusieurs ports d'adresses des mémoires de données en fonction de leurs bits de poids fort. Ce module propose donc un mode de diffusion (*broadcast*) qui permet de présenter une même adresse sur plusieurs ports simultanément et de limiter le nombre de générateurs à programmer et à activer.

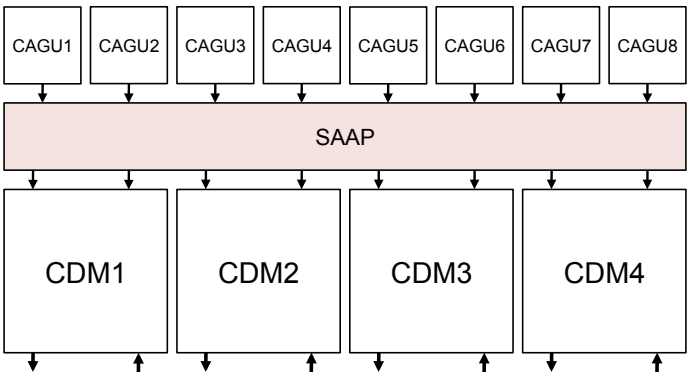


Fig. 2.16: *Environnement du module interconnectant les générateurs d'adresses d'un cluster aux mémoires de données simple double-port.*

L'insertion de ce composant entraîne une modification légère du jeu d'instructions des AGU intra *cluster*. Concrètement, les instructions OUT et ROP disposent d'un argument (ou masque) dont chaque élément binaire est associé à une sortie particulière de l'interface. Ainsi, si le bit $b_i = 1$ alors l'adresse générée par l'AGU est transmise au port d'adresses correspondant. Par conséquent, la taille

des instructions supportées par les générateurs intra *cluster* est déterminée par celle de l'instruction ROP et est égale à $N_{\text{adresse}} + N_{\text{masque}} + 6$ bits.

L'architecture de l'interface SAAP peut être définie selon deux approches. La première approche consiste à autoriser l'interconnexion de chaque sortie d'un générateur vers tous les ports d'adresses des mémoires de données présentes. Cette solution garantit donc un degré de flexibilité maximale mais se revèle être la plus gourmande en surface tant du point de vue de l'interface que de celui des générateurs d'adresses où la largeur des instructions mémorisées, et donc la surface de la mémoire correspondante, est maximale et est égale à $N_{\text{adresse}} + 2 \cdot N_{\text{mémoires}} + 6$ bits. La deuxième approche, quant à elle, exploite la symétrie des deux ports d'adresses des mémoires de données dans le but de minimiser la surface occupée par l'interface sans pour autant dégrader de manière significative sa flexibilité. Plus précisément, chaque sortie des générateurs d'adresses est connectée à l'un des deux ports d'adresses des mémoires de données, à savoir les AGU dites paires (respectivement impaires) peuvent être connectées au premier (respectivement second) port d'adresses de chaque mémoire. Dans ce cas de figure, mise en œuvre dans le cadre de l'élaboration de MOREA, la surface de l'interface, bien que représentant une surface cumulée inférieure à 1% de celle d'une tuile, est réduite de 39% par rapport à celle relative à la première approche. D'autre part, la largeur des instructions des générateurs d'adresses intra *cluster* est désormais de $N_{\text{adresse}} + N_{\text{mémoires}} + 6$ bits ce qui nécessite une mémoire d'instructions de moindre capacité et donc plus efficace sur le plan énergétique.

2.6 Structure du contrôleur de reconfiguration

Dans une tuile, la programmation des ressources de contrôle, à savoir notamment les générateurs d'adresses, et la reconfiguration dynamique des ressources opératives (unités fonctionnelles et le réseau d'interconnexions) sont gérées par une unité de contrôle hiérarchisée. Au plus haut niveau, un contrôleur de tuile est donc chargé du suivi de la programmation des générateurs d'adresses de la tuile et des contrôleurs de *cluster*, et de l'activation et la synchronisation du générateur d'adresses de la mémoire de la tuile et des contrôleurs de *cluster*. En outre, il s'occupe de la reconfiguration du réseau *crossbar*. Au sein d'un *cluster*, une unité de contrôle commande l'activation et la synchronisation des générateurs d'adresses ainsi que la reconfiguration de l'interconnexion multibus et des unités fonctionnelles. Les paragraphes suivants s'attachent donc à présenter essentiellement le jeu d'instructions de ces deux types de composant.

2.6.1 Description du contrôleur de tuile

Le contrôleur de tuile est un processeur programmable fondé sur le paradigme RISC. De la même façon que le chemin de données de la tuile traite des données signées sur 32 bits, l'unité de calcul du contrôleur manipule également des valeurs positives et négatives codées sur 32 bits. Le jeu d'instructions du contrôleur est constitué de 16 instructions de taille 32 bits (tableau 2.6). Celui-ci est sensiblement comparable à celui des AGU à quelques différences près toutefois. Ainsi, l'instruction LOAD ne permet de modifier dans un registre que les 16 bits de poids faible. D'autre part, une instruction NOC autorise des communications ou l'envoi de requêtes particulières vers l'interface de la tuile avec le réseau NoC. Concrètement, cette instruction peut permettre de transférer depuis des

ressources de mémorisation externes vers celles de la tuile des instructions et des données. Dans ce cas, un drapeau généré par l'interface doit être positionné à 1 à la fin du transfert. Finalement, l'argument de cette instruction, codé sur 28 bits, est déterminé en fonction des paramétrages possibles de l'interface avec le NoC.

Code opératoire	**Instruction**	**Syntaxe**	**Description**
1	LOAD	Rn, < opérande >	Donnée immédiate → Rn
2	GET	Rn	Donnée externe → Rn
3	ADD	Rx, Ry, Rz	Ry + Rz → Rx
4	SUB	Rx, Ry, Rz	Ry − Rz → Rx
5	AND	Rx, Ry, Rz	Ry · Rz → Rx
6	ASH	Rx, Ry, $\pm d$	*Arithmetic Shift*
7	NOC	< requête >	
8	CONF	< cible >, < configuration >	
9	RUN	< masque >	
10	BNZ	< étiquette >	*Branch if Not Zero*
11	BNG	< étiquette >	*Branch if Negative*
12	BXF	< étiquette >	*Branch if External Flag Set*
13	BRA	< étiquette >	*Branch Always*
14	WAIT	(FLAG), < N_{cycles} ou masque >	*Wait State*
15	END		Programme terminé
0	NOP		*No Operation*

Tab. 2.6: *Description du jeu d'instructions du contrôleur de tuile.*

Par ailleurs, une instruction CONF permet de programmer notamment l'interconnexion *crossbar* entre les quatre *clusters*. Le champ *cible* précise alors la nature des multiplexeurs considérés (par exemple ceux en entrée de l'un des quatre *clusters*) dont les bits de sélection sont définis par le champ < configuration >. L'instruction RUN, quant à elle, active par le biais du champ < masque > une ou plusieurs unités programmables dont le générateur d'adresses de la mémoire de la tuile et les quatre contrôleurs de *cluster*. À la suite de quoi, l'instruction WAIT permet, entre autre et par l'entremise de l'option FLAG, d'attendre en scrutant l'affirmation du drapeau *Fin* des unités programmables précitées ainsi que celui émis par l'interface avec le NoC.

Enfin, les applications de traitement du signal et de l'image peuvent présenter un comportement dynamique, c'est à dire dépendant des données traitées. De ce fait, le jeu d'instructions du contrôleur de tuile et aussi celui des contrôleurs de *cluster* comportent des directives GET et BXF permettant une interaction avec le chemin de données de la tuile.

2.6.2 Description du contrôleur de *cluster*

L'architecture du contrôleur de *cluster* et son jeu d'instructions sont quasiment similaires à ceux du contrôleur de tuile. Cependant, les instructions NOC et CONF sont remplacées par deux autres directives, à savoir CSW et CHW, permettant de reconfigurer les ressources opératives et l'interconnexion du *cluster* en fonction des types de traitements rencontrés dans les applications multimédia (tableau 2.7). Plus précisément, celles-ci présentent de manière générale des traitements dits réguliers où les données sont manipulées via des opérations répétitives et présentant bien souvent des degrés

de parallélisme d'opérations et de données importants. Puis, les valeurs extraites de ces traitements sont analysées par le biais de fonctions irrégulières où des opérations diverses et variées se succèdent rapidement.

Code opératoire	**Instruction**	**Syntaxe**	**Description**
1	LOAD	Rn, < opérande >	Donnée immédiate → Rn
2	GET	Rn	Donnée externe → Rn
3	ADD	Rx, Ry, Rz	Ry + Rz → Rx
4	SUB	Rx, Ry, Rz	Ry − Rz → Rx
5	AND	Rx, Ry, Rz	Ry · Rz → Rx
6	ASH	Rx, Ry, $\pm d$	*Arithmetic Shift*
8	CSW	< sélection >, < configuration >	Configuration *Software*
8	CHW	< cible >, < configuration >	Configuration *Hardware*
9	RUN	< masque >	
10	BNZ	< étiquette >	*Branch if Not Zero*
11	BNG	< étiquette >	*Branch if Negative*
12	BXF	< étiquette >	*Branch if External Flag Set*
13	BRA	< étiquette >	*Branch Always*
14	WAIT	(FLAG), < N_{cycles} ou masque >	*Wait State*
15	END		Programme terminé
0	NOP		*No Operation*

Tab. 2.7: *Description du jeu d'instructions d'un contrôleur de clusters.*

Pour implémenter les traitements irréguliers, l'instruction CSW autorise une reconfiguration rapide, c'est à dire en un cycle, des ressources du *cluster* et permet donc de mettre en œuvre de manière relativement efficace, selon une implémetation temporelle ou logicielle, un traitement particulier. Concrètement, cette instruction considère le chemin de données du *cluster* comme étant celui d'un processeur VLIW à quatre voies et, dans un souci de réduction du volume de données de configuration, n'implémente que des motifs de calcul de type *Read, Modify and Write* (RMW) :

1. étape *Read* où une unité fonctionnelle i lit ses opérandes dans la mémoire i et une mémoire j, avec $j = i$ ou $j \neq i$
2. étape *Modify* où l'unité fonctionnelle i effectue une opération arithmétique ou logique et, le cas échéant, un décalage sur l'une des données en entrée ou sur le résultat en sortie
3. étape *Write* où le résultat de l'opération réalisée par l'unité fonctionnelle i est placée dans la mémoire i.

Dans cette instruction, le champ < cible > permet d'activer ou de désactiver une ou plusieurs voies en contrôlant l'état des registres disposés en entrée et en sortie des unités fonctionnelles. Ainsi, dans le cas de la désactivation d'une unité fonctionnelle, la sortie de ses registres est figée et l'opérateur nengendre donc pas de consommation dynamique.

À l'opposé, l'instruction CHW permet d'exploiter toute la flexibilité des ressources du *cluster* et donc d'implémenter de manière spatiale ou matérielle des traitements présentant des calculs répétitifs. Finalement, l'instruction RUN permet d'activer un ou plusieurs générateurs d'adresses intra *cluster*.

2.7 Synthèse

Dans ce chapitre, l'architecture de MOREA a été présentée. Préalablement, le modèle d'application que nous considérons a été posé. Il est fondé notamment sur le découpage successif d'une application en processus puis en tâches. À partir de ce modèle, nous avons dégagé l'organisation générale de MOREA. Globalement, celle-ci est structurée autour d'un réseau programmable hiérarchisé qui interconnecte l'ensemble des ressources de calcul et de mémorisation disponibles. De ce fait, il permet d'implémenter une grande diversité de chemins de données entre les unités opératives et de stockage tout en minimisant le coût en surface, délai de propagation et consommation de l'interconnexion.

Dans MOREA, les traitements sont supportés, en premier lieu, par des unités fonctionnelles extraites de l'architecture reconfigurable DART [12]. Dès lors, ces dernières sollicitent, par l'entremise du réseau d'interconnexions, une unité mémoire dont la structure a été définie en fonction des besoins exprimés par les applications de traitement du signal et de l'image, en termes notamment de variété des motifs d'accès mémoire et des séquences d'adresses pour l'unité de génération d'adresses, mais également en tenant compte des contraintes de conception inhérentes au domaine de l'embarqué, à savoir les contraintes de surface, de performances et de dissipation énergétique. Finalement, le contrôle de la reconfiguration des ressources de MOREA est réalisé par une unité *ad-hoc* qui est capable, en outre, de supporter le traitement d'applications dynamiques, c'est-à-dire dont le comportement est dépendant des données traitées par l'application.

Dans le chapitre suivant, nous allons caractériser l'architecture de MOREA en surface, performances et consommation. Les données obtenues permettront alors de comparer notre solution à l'architecture reconfigurable DART notamment et de démontrer ainsi l'intérêt de notre concept de hiérarchie mémoire reconfigurable. Ensuite, nous validerons notre proposition architecturale, c'est-à-dire que nous démontrerons la capacité de MOREA à mettre en œuvre des applications multimédia. Cette section sera également l'occasion d'analyser plus précisément les atouts de notre unité de génération d'adresses.

Chapitre 3

Caractérisation et validation de MOREA

3.1 Caractérisation de MOREA

Dans le cadre de cette thèse, un modèle VHDL d'une tuile complète de MOREA a été développé. Précisément, les différents éléments d'une tuile, à l'exception des mémoires de données et d'instructions dont la description a été générée par des compilateurs propriétaires, ont été décrits en VHDL, puis simulés et validés au niveau fonctionnel et RTL sous Mentor Graphics ModelSim. À la suite de quoi, chaque élément a été synthétisé grâce à l'outil Synopsys Design Compiler et à partir d'une bibliothèque ASIC 130 nm. Ainsi, les résultats présentés dans ce chapitre sont valables pour cette technologie uniquement. De nouveau, chaque bloc a été simulé et validé au niveau porte via ModelSim. Par ailleurs, les estimations en consommation ont été effectuées grâce à l'outil Synopsys PrimeTime PX et à partir de fichiers d'activité produits par ModelSim.

Dans cette section, en premier lieu, nous caractérisons en surface, performances et consommation les éléments d'un *cluster* de MOREA. Puis, nous évaluons les caractéristiques d'une tuile et comparons ses performances avec celles de l'architecture DART notamment. Cette étape permettra alors de mettre en évidence l'intérêt de notre proposition de hiérarchie mémoire reconfigurable.

3.1.1 Évaluation des performances d'un *cluster* de MOREA

La figure 3.1 rappelle l'architecture d'un *cluster* de MOREA. Dans un *cluster*, les traitements sont supportés par quatre unités fonctionnelles (UF_i), dont deux multiplieurs-additionneurs (UF1 et UF3) et deux ALU (UF2 et UF4), qui réalisent chacune une opération sur deux données 32 bits (16 bits dans le cas d'une multiplication) et produisent un résultat sur 32 bits. Ces unités sont interconnectées à quatre mémoires simple double-port (*Cluster Data Memory*, CDM_i), c'est-à-dire disposant d'un port de lecture et d'un port d'écriture, de capacité 512×32 bits, grâce à un motif flexible de type multi-bus. Ce dernier communique avec le réseau *crossbar* de la tuile par le biais de quatre entrées et quatre sorties. Au sein d'un *cluster*, les accès aux données mémorisées sont contrôlés par huit générateurs d'adresses (*Cluster Address Generation Unit*, $CAGU_i$) connectés aux quatre mémoires par l'entremise d'une interface auto-adaptable (*Self Adaptive Address Path*, SAAP). Chaque AGU dispose d'une mémoire d'instructions de dimensions 64×19 bits. Enfin, le contrôle de la reconfiguration du chemin de données du *cluster*, de même que l'activation des AGU, sont réalisées

par une unité programmable (*Cluster Controller*, CCTRL). Pour ce faire, celle-ci est équipée d'une mémoire d'instructions de capacité 128×32 bits.

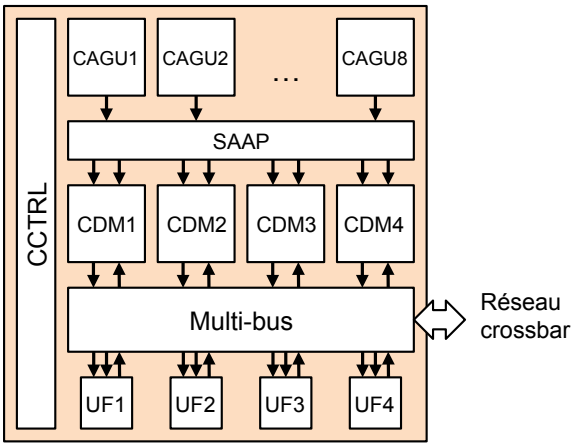


Fig. 3.1: *Architecture d'un cluster de MOREA.*

Le tableau 3.2 présente la performance, c'est-à-dire le temps du chemin critique (t_{cc}), la surface et la consommation des différents éléments d'un *cluster*. Ainsi, de manière globale, la fréquence de fonctionnement de la tuile, déterminée à $F_{clock} = 192$ MHz, correspond à la période d'horloge des unités programmables (AGU et contrôleurs de configuration). Cet état de fait s'explique par les temps d'accès importants des mémoires d'instructions utilisées. En effet, leur description provient d'un compilateur qui optimise les mémoires en surface prioritairement. Les données en surface et en consommation précisées dans ce tableau ont été estimées à $F_{clock} = 192$ MHz.

Unité	$N_{\textbf{instances}}$	t_{cc} **(ns)**	**Surface** (μm^2)	**Consommation (mW)**
UF1 / UF3	2	3,6	42 422	5,85
UF2 / UF4	2	4,1	37 058	7,71
Multi-bus	1	2,7	116 158	25,58
CDM	4	1,3	198 988	34,83
SAAP	1	1,4	11 655	2,38
CAGU	8	5	31 095	4,6
CCTRL	1	5,1	106 571	12,08
Cluster	-	5,2	1 438 056	243,28

Tab. 3.1: *Performance, surface et consommation d'énergie des éléments d'un cluster de MOREA.*

La puissance de calcul crête développée par un *cluster* de MOREA est donc estimée à 768 MOPS. Cette valeur a été obtenue en multipliant la fréquence d'horloge du système, et donc le nombre d'opérations par seconde réalisées par une unité fonctionnelle, par le nombre d'unités fonctionnelles présentes dans un *cluster*. Par ailleurs, afin de quantifier la performance de l'unité mémoire de MOREA,

la métrique MAPS[1] a été définie. Celle-ci représente le nombre maximum de lectures et d'écritures (ou d'accès) par seconde que peut réaliser une unité fonctionnelle. Ainsi, dans un *cluster*, un opérateur peut effectuer $3 \times F_{\text{clock}}$ millions d'accès par seconde, soit 576 MAPS. Dans la suite de cette section, nous distinguerons en outre les MAPS dits locaux ou LMAPS, qui caractérisent les accès effectués par un eunité fonctionnelle aux ressources mémoire du *cluster* auquel elle appartient, des MAPS dits globaux ou GMAPS, qui correspondent aux accès effectués par une unité fonctionnelle aux ressources mémoire d'un autre *cluster*.

Ainsi, comparativement à un DPR de DART (cf. figure 2.6), les performances offertes par un *cluster* de MOREA sont relativement identiques, que ce soit en termes de puissance de calcul que de nombre d'accès mémoire. Cependant, l'architecture optimisée des générateurs d'adresses de MOREA, intégrant un accélérateur matériel pour la mise en œuvre des séquences les plus courantes, permet d'atteindre plus fréquemment des niveaux de performances maximales contrairement au cas de l'utilisation du processeur DART. Par ailleurs, comparativement à une IP dédiée proposant un niveau de performances identique, le surcoût de MOREA en termes de surface et de consommation est principalement dû au réseau multi-bus, qui représente 8% de la surface du *cluster* et est à l'origine de 11% de sa consommation, et au contrôleur de reconfiguration, qui représente 7% de la surface du *cluster* et est à l'origine de 5% de sa consommation. On notera ainsi que le coût du réseau d'interconnexions, tant en surface qu'en consommation, est plutôt négligeable comparé au coût de l'interconnexion dans les composants FPGA.

Finalement, si les performances d'un DPR de DART et d'un *cluster* de MOREA sont proches, il en est en revanche tout autre au niveau de l'architecture globale de DART et de MOREA. La suite de cette section précise cette remarque en étudiant les caractéristiques d'une tuile complète.

3.1.2 Évaluation des performances de la tuile de MOREA

La figure 3.2 rappelle l'architecture d'une tuile de MOREA. Celle-ci est donc constituée de quatre *clusters* qui s'échangent des données par le biais d'un réseau *crossbar*. Ce réseau permet en outre aux *clusters* d'accéder à une mémoire de données simple-port (*Tile Data Memory*, TDM) de capacité $16K \times 32$ bits. Les accès à cette mémoire sont gérés par un générateur d'adresses programmable comprenant une mémoire d'instructions de dimensions 128×21 bits. Finalement, le contrôle de la tuile est dévolue à une unité spécifique programmable (*Tile Controller*, TCTRL) intégrant une mémoire d'instructions de capacité 256×32 bits.

La puissance de calcul crête d'une tuile de MOREA est donc de 3072 MOPS. À propos de son unité de stockage, les MAPS globaux (GMAPS) sont équivalents aux MAPS locaux (LMAPS), soit 576 GMAPS, à une latence près toutefois. Ce délai est de 3 cycles (un délai pour lire une donnée et traverser le réseau multi-bus du premier *cluster*, un cycle pour traverser le réseau *crossbar* et un dernier cycle pour traverser le réseau multi-bus du deuxième *cluster* et atteindre l'unité fonctionnelle). Dans le processeur DART, le réseau segmenté est utilisé pour chaîner des unités fonctionnelles appartenant à des DPR distincts. Ainsi, la performance de l'unité mémoire de cette architecture est

[1]MAPS : Millions d'Accès Par Seconde

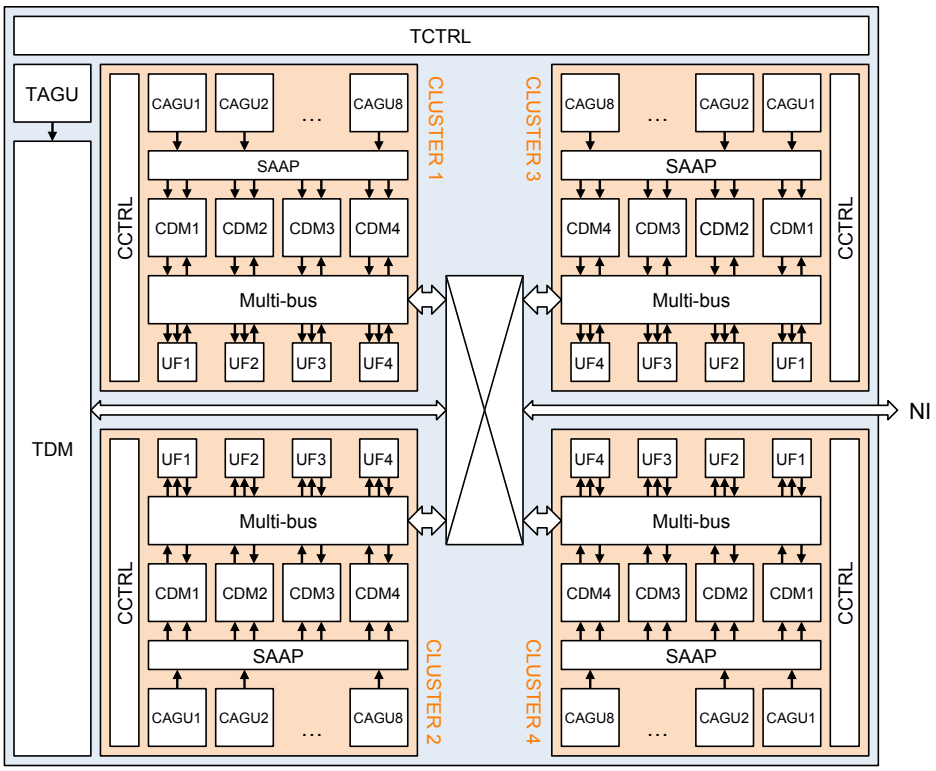


FIG. 3.2: *Architecture d'une tuile de MOREA.*

Unité	$N_{\textbf{instances}}$	t_{cc} **(ns)**	**Surface** (μm^2)	**Consommation (mW)**
Cluster	4	5,1	1 438 056	243,28
crossbar	1	1,7	140 322	23,96
TDM	1	2,9	1 685 665	45,03
TAGU	1	4,9	71 714	8,31
TCTRL	1	5,2	117 886	12,41
Tuile	-	5,2	7 767 811	1062,83

TAB. 3.2: *Performance, surface et consommation d'énergie des éléments d'une tuile de MOREA.*

de 128 GMAPS. Ceci s'explique par le fait qu'une unité fonctionnelle appartenant à un DPR ne peut accéder via le réseau segmenté à une donnée présente dans l'unité mémoire d'un autre DPR. Cette donnée doit forcément passer par le deuxième niveau mémoire de DART engendrant donc des délais supplémentaires. En revanche, dans MOREA, ce deuxième niveau est court-circuité grâce à la présence du réseau *crossbar* dont l'impact sur la surface et la consommation d'une tuile est négligeable.

3.2 Validation de MOREA

Cette section va permettre de valider notre proposition d'architecture reconfigurable dynamiquement, c'est à dire qu'elle va démontrer la capacité de MOREA de pouvoir supporter des applications de traitement du signal et de l'image. Dans ce but, nous allons y implémenter une application de

compression vidéo respectant la norme H.264/MPEG-4 AVC. En effet, celle-ci induit des traitements réguliers exhibant notamment des séquences d'adresses affines et également dépendantes des données manipulées, ainsi que des traitements flot de contrôle reposant sur une analyse des résultats précédemment obtenus et une prise de décision en conséquence concernant les opérations à réaliser. D'autre part, cette phase d'implémentation va permettre de caractériser les performances de notre unité de génération d'adresses en matière de vitesse de calcul et de consommation d'énergie.

3.2.1 Description d'un encodeur vidéo H.264/MPEG 4 AVC

Les systèmes de télécommunications actuels transmettent des informations de différentes natures, à savoir du texte, de la parole et de la musique, ainsi que des images et de la vidéo. Dans ce dernier cas, le volume de données transférées est conséquent. En effet, une séquence vidéo comporte au minimum 25 images par seconde (contrainte imposée par le phénomène de persistance rétinienne), une image est composée, d'après les derniers formats en vigueur, de plusieurs dizaines de milliers de pixels qui sont codés chacun sur une dizaine de bits environ. Par voie de fait, le transfert efficace de ce type de média nécessite de réduire sa quantité de données. Dans un système de télécommunications, au niveau de l'unité émettrice, cette fonction est dévolue à un encodeur vidéo qui traduit une certaine séquence dans un format nécessitant un volume de données moins important. Dès lors, au niveau de l'unité réceptrice, un décodeur est requis afin de décompresser la séquence émise et donc de pouvoir lire la vidéo originale. De manière générale, une application de compression vidéo est structurée autour de trois fonctions principales [51] (figure 3.3) :

1. le modèle temporel,
2. le modèle spatial,
3. le codeur entropique.

La suite de cette première partie détaille le fonctionnement notamment des deux premiers traitements.

3.2.1.1 Description du modèle temporel

Globalement, une séquence vidéo est constituée d'une succession d'images ou trames (*Frames*) relativement similaires. Plus précisément, les pixels d'une trame F_{n+1} sont identiques à ceux d'une trame antérieure F_n à un déplacemet près. Dans un encodeur vidéo, le modèle temporel a pour fonction de compresser la trame F_{n+1} dite courante en la codant par la valeur du déplacement de ses pixels par rapport à ceux de la trame F_n dite de référence. Ainsi, en entrée de ce bloc, sont présentées les trames F_n et F_{n+1} (figure 3.3). Au sein de ce bloc, une fonction dite d'estimation de mouvement calcule alors le déplacement entre les deux images F_n et F_{n+1}, et génère un résultat donné sous la forme d'un vecteur de mouvement. En outre, une fonction appelée compensation de mouvement définit un résidu qui représente l'erreur de reconstruction de la trame F_{n+1} à partir de la trame F_n.

Dans la réalité, tous les pixels de F_n ne subissent pas exactement la même translation permettant d'obtenir l'image F_{n+1}. Par conséquent, dans un souci de compromis acceptable entre le résultat visuel obtenu après décodage et le temps de calcul nécessaire à la compression de la séquence vidéo, la

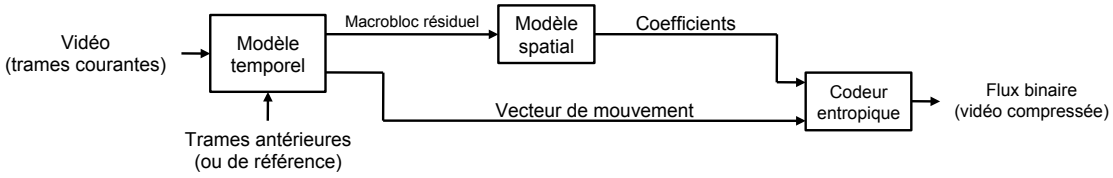


FIG. 3.3: *Schéma-bloc général d'un encodeur vidéo.*

trame courante F_{n+1} est préalablement partitionnée en plusieurs macroblocs (MB) de dimensions en général 16×16 pixels. Puis, pour chaque macrobloc courant, la fonction d'estimation de mouvement détermine son déplacement en recherchant dans la trame de référence le macrobloc qui lui ressemble le plus au sens de la minimisation d'un certain critère de ressemblance. Le vecteur de mouvement associé au macrobloc courant considéré est donc obtenu par la différence entre la position de ce dernier et celle du macrobloc de référence le plus ressemblant. De surcroît, la fonction de compensation de mouvement génère un macrobloc résiduel en soustrayant au macrobloc courant celui de référence le plus ressemblant. Finalement, dans un encodeur vidéo, le modèle temporel détermine un ensemble de vecteurs et de résidus associés aux différents macroblocs de la trame courante.

La définition du meilleur macrobloc de référence est fondée sur la minimisation d'un critère de ressemblance particulier. Dans la littérature, divers critères ont été proposés jusqu'à présent tels que par exemple [51] :

– *Mean Squared Error* :

$$\mathrm{MSE} = \frac{1}{16^2} \sum_{i=0}^{15} \sum_{j=0}^{15} (C_{ij} - R_{ij})^2 \tag{3.1}$$

– *Mean Absolute Error* :

$$\mathrm{MAE} = \frac{1}{16^2} \sum_{i=0}^{15} \sum_{j=0}^{15} |C_{ij} - R_{ij}| \tag{3.2}$$

– *Sum of Absolute Differences* :

$$\mathrm{SAD} = \sum_{i=0}^{15} \sum_{j=0}^{15} |C_{ij} - R_{ij}| \tag{3.3}$$

avec C_{ij} (espectivement R_{ij}) le pixel de coordonnées (i, j) du macrobloc courant (respectivement de référence). Généralement, de part sa relative simplicité de mise en œuvre, le critère SAD est celui le plus fréquemment adopté dans les encodeurs vidéo. Aussi, dans le cadre de la validation de MOREA, nous adopterons également ce critère pour l'implémentation de la tâche d'estimation de mouvement.

Dans une séquence vidéo, le déplacement entre deux trames successives est généralement de faible amplitude. De ce fait, la recherche du meilleur macrobloc de référence est limitée à une zone ou fénêtre centrée sur la position relative du macrobloc courant considéré. Par ailleurs, la détermination du macrobloc le plus ressemblant peut suivre différents algorithmes. De manière globale, ceux-ci peuvent être répartis en deux catégories :

1. les algorithmes dits exhaustifs qui, pour une fenêtre de dimensions $W \times W$ pixels considèrent W^2 macroblocs de référence candidat. Ces algorithmes procurent donc le meilleur résultat visuel possible mais demandent une puissance de calcul importante afin de rechercher rapidement le meilleur macrobloc de référence parmi les W^2 candidats possibles.
2. les algorithmes dits rapides qui requièrent un temps de calcul moindre par rapport aux approches exhaustives mais qui procurent en contrepartie un résultat visuel de qualité inférieure. À titre d'exemples, nous pouvons citer les algorithmes de recherche en diamant ou ceux de recherche logarithmique.

Dans la famille des algorithmes rapides, un de ceux les plus populaires est l'algorithme de recherche en trois étapes qui fournit un compromis intéressant entre la qualité du résultat visuel engendré et le temps de calcul [52]. Concrètement, la première étape de cet algorithme consiste à calculer le critère de ressemblance pour la macrobloc de référence situé à la même position que celui courant ainsi que pour huit autres candidats disposés à une distance d du macrobloc courant (figure 3.4). À la fin de cette première étape, la zone de recherche est déplacée autour de la position du meilleur macrobloc de cette étape et la distance d est réduite de moitié. Dès lors, la procédure ci-décrite est répétée pour la deuxième et la troisième étape. Ainsi, en considérant $W = 15$ et $d = 4$, l'algorithme de recherche en trois étapes ne nécessite que 25 calculs du critère de ressemblance au lieu des 225 calculs imposés par une méthode exhaustive.

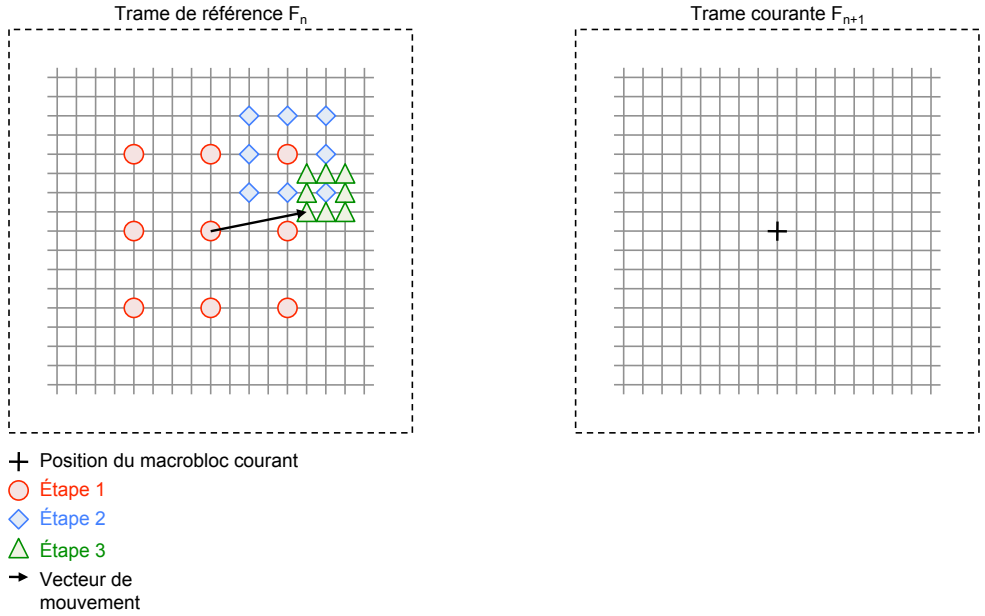


FIG. 3.4: *Description de l'algorithme de recherche en trois étapes.*

L'annexe F présente la description C de la fonction d'estimation de mouvement reposant sur un algorithme de recherche en trois étapes. Cette fonction prend donc comme argument en entrée un pointeur vers un macrobloc courant et une fenêtre de recherche, la position d'un macrobloc de référence par rapport à celle du macrobloc courant, ainsi qu'un pointeur vers la valeur minimale

courante du critère de ressemblance. Dès lors, elle met à jour la valeur du vecteur de mouvement et celle minimale du critère de ressemblance. Globalement, cette fonction est composée de deux phases, à savoir un premier traitement régulier visant à déterminer la valeur du critère pour le macrobloc de référence considéré et induisant notamment une séquence d'adresses affines pour la lecture du macrobloc courant et une autre dépendante de la position du macrobloc de référence considéré dans le cadre de l'exploration de la fenêtre de recherche, et un deuxième traitement irrégulier qui repose, d'une part, sur la comparaison de la valeur calculée du critère avec sa valeur minimale courante et, d'autre part, sur la mise à jour conditionnelle de cette valeur minimale et du vecteur de mouvement avec la position du macrobloc de référence considéré.

Suite à l'estimation de mouvement, un traitement de compensation de mouvement soustrait au macrobloc courant celui de référence le plus ressemblant. Il en résulte alors un macrobloc résiduel représentant l'erreur d'estimation du macrobloc à partir de celui de référence. Ce résidu est compressé par le biais du modèle spatial que nous présentons ci-après.

3.2.1.2 Description du modèle spatial

Dans une image, les pixels adjacents sont généralement de la même couleur. Cette redondance spatiale se traduit, dans le domaine fréquentiel, par une concentration de l'énergie de l'image dans les basses fréquences, l'énergie portée par les hautes fréquences étant faible voire quasiment nulle (figure 3.5). Dans un encodeur vidéo, le modèle spatial exploite cette propriété afin d'optimiser, suite à une étape de quantification, le gain en compression du codeur entropique. Concrètement, le macrobloc résiduel provenant du modèle temporel est représenté dans le domaine fréquentiel grâce à l'application d'une transformée en cosinus discrète dans un espace à deux dimensions (DCT[2] 2D).

Pour un macrobloc X de dimensions $N \times N$ pixels, sa représentation dans le domaine fréquentiel selon les axes u et v via la DCT 2D est donnée par le formule brute 3.4.

$$y_{uv} = \frac{2}{N} \cdot g(u) \cdot g(v) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x_{ij} \cdot \cos\left[(i + \frac{1}{2}) \cdot \frac{u\pi}{N}\right] \cdot \cos\left[(j + \frac{1}{2}) \cdot \frac{v\pi}{N}\right] \tag{3.4}$$

où

$$g(i) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } i = 0 \\ 0 & \text{sinon} \end{cases}$$

D'autre part, une DCT 2D peut être également décrite sous la forme d'un double produit matriciel (équation 3.5). Plus précisément, en exploitant la propriété d'associativité du produit matriciel, une DCT 2D est équivalente à deux séries successives de DCT 1D appliquées, premièrement, sur les lignes du macrobloc original (DCT_{1x}, figure 3.6) et, deuxièmement, sur les colonnes du résultat

[2]DCT : *Discrete Cosine Transform*

(a) (b)

Fig. 3.5: *Représentation d'une image codée sur 256 niveaux de gris (a) dans le domaine fréquentiel (b) grâce à l'application de la DCT 2D. Les composantes fréquentielles d'amplitudes élevées (respectivement de faibles amplitudes) sont signalées par des pixels dont la couleur tend vers le blanc (respectivement le noir). Ainsi, on observe que l'énergie de l'image est principalement portée par les basses fréquences représentées en haut et à gauche de la seconde image.*

intermédiaire (DCT_{1y}, figure 3.6).

$$Y = H \cdot X \cdot H^{\mathrm{T}} = (H \cdot (X \cdot H^{\mathrm{T}})) \tag{3.5}$$

où

$$h_{ij} = g(i) \cdot \sqrt{\frac{2}{N}} \cdot \cos\left[(i + \frac{1}{2}) \cdot \frac{j\pi}{N}\right]$$

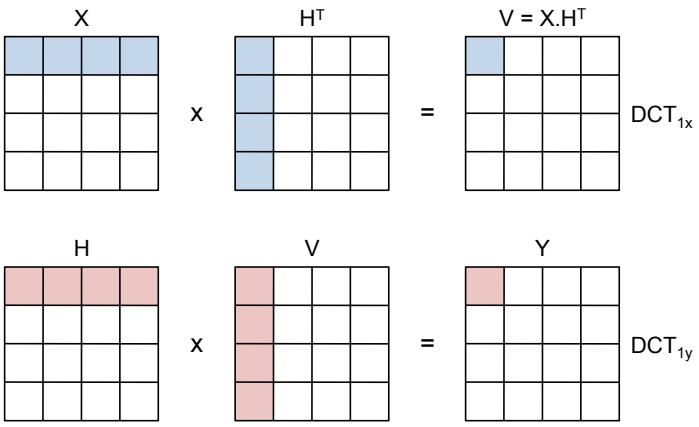


Fig. 3.6: *Représentation matricielle de la DCT 2D appliquée à un macrobloc de dimensions* 4×4 *pixels.*

Dans le cas de la norme H.264/MPEG-4 AVC, la DCT 2D est appliquée à des macroblocs de dimensions 4×4 pixels. De plus, les coefficients h_{ij}, de part la présence dans leur définition d'une racine carrée de 2 (équation 3.5), sont irrationnels. Cette particularité implique qua la qualité de la transformée directe combinée à la transformée inverse est fortement dépendante du support matériel d'exécution, c'est à dire du nombre de bits disponibles pour coder les coefficients. Aussi, pour découpler l'efficacité de la transformée en cosinus discrète et du matériel, la norme H.264/MPEG-4 AVC préconise l'usage d'une approximation entière de la DCT 1D dont le graphe flot de données induit des multiplications par 2 qui peuvent être substituées par des décalages élémentaires à gauche [53] (figure 3.7). Par conséquent, la mise en œuvre de la DCT 2D consiste à appliquer le graphe de la figure **??**, dans un premier temps, aux lignes du macrobloc source puis, aux colonnes du résultat intermédiaire.

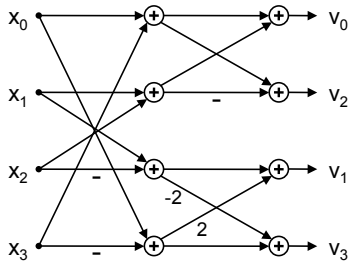


Fig. 3.7: *Graphe flot de données de l'approximation entière de la DCT 1D dans le cadre de la norme de compression vidéo H.264/MPEG-4 AVC.*

Ainsi, pour chaque macrobloc résiduel produit par le modèle temporel, le modèle spatiel détermine 16 matrices de 4×4 coefficients. Dès lors, chaque matrice subit une étape de quantification qui a pour effet notamment d'annuler les valeurs correspondant aux hautes fréquences. Pour une matrice donnée, les coefficients sont ensuite lus selon un motif zigzag et la chaîne binaire résultante est concaténée au vecteur de mouvement et transmise au codeur entropique. Ce dernier exploite la redondance statistique de la chaîne binaire en entrée, c'est à dire que les suites de bits les plus fréquemment observées sont codées avec des mots courts. Le flux binaire ainsi produit représente donc de manière minimale la trame courante. Finalement, cette fonction repose sur des opérations au niveau bit, pouvant être implémentée par exemple par des LUT, et est typiquement mise en œuvre par des structures à grain fin. De ce fait, la suite de ce troisième chapitre considère uniquement la réalisation sur MOREA du modèle temporel et du modèle spatial.

3.2.2 Implémentation de l'encodeur vidéo H.264/MPEG-4 AVC sur une tuile de MOREA

La dernière partie de ce chapitre décrit l'implémentation de l'encodeur vidéo H.264/MPEG-4 AVC sur une tuile de MOREA. Pour ce faire, tout d'abord, nous modélisons cette application sous la forme d'un graphe de tâches communiquant par le biais de tampons de données. Ensuite, nous expliquons la mise en œuvre de chaque tâche sur une tuile de MOREA. Nous profitons également de cette étape pour évaluer l'impact de l'architecture des AGU sur les performances de MOREA.

Finalement, nous discutons de l'enchaînement des différentes tâches mises en œuvre en explicitant les mécanismes de contrôle induits.

3.2.2.1 Description de l'implémentation de l'estimation de mouvement

L'application de compression vidéo est modélisée par un graphe constitué de trois tâches successives (figure 3.8). La première tâche est l'estimation de mouvement et consomme des données dans deux tampons contenant respectivement le macrobloc courant et la fenêtre de recherche. Elle produit un vecteur de mouvement qui est placé dans un troisième tampon. Suite à l'écriture du résultat de l'estimation de mouvement dans le troisième tampon, la deuxième tâche est lancée. Celle-ci est la compensation de mouvement qui exploite le résultat de la tâche précédente afin de définir l'erreur d'estimation représentée sous la forme d'un macrobloc résiduel. La production de ce résidu dans le tampon t_4 permet alors d'amorcer la tâche finale. Cette dernière applique une transformée en cosinus discrète sur ce résidu et produit une matrice de coefficients.

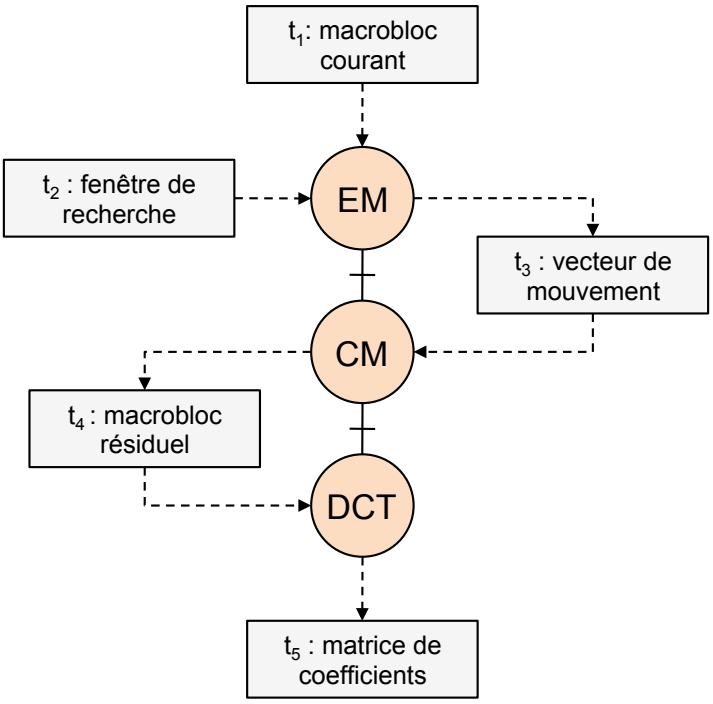


Fig. 3.8: *Modélisation sous la forme d'un graphe de tâches de l'application de compression vidéo H.264/MPEG-4 AVC. Ce graphe est composé de trois tâches : la tâche d'estimation de mouvement (EM) qui consomme ses données dans les tampons t_1 et t_2, contenant respectivement le macrobloc courant et la fenêtre de recherche, et produit un vecteur de mouvement placé dans un troisième tampon t_3 ; la tâche de compensation de mouvement (CM) qui lit également les tampons t_1 et t_2 (les flèches correspondantes ne sont pas indiquées sur la figure ci-dessus) et écrit dans le tampon t_4 où est disposé le macrobloc résiduel ; enfin, la tâche de la DCT 2D qui applique une transformée en cosinus discrète aux données du tampon t_4 et produit une matrice de coefficients stockée dans un cinquième et dernier tampon t_5.*

Le temps d'exécution de l'estimation de mouvement est dominé par le temps d'évaluation du critère de ressemblance contenu dans l'estimation de mouvement. Ce dernier, dans le cas présent, impose la mise en œuvre d'un motif SAD qui repose sur trois opérations (une soustraction, une

valeur absolue et une accumulation). Dans les quatre *clusters* d'une tuile, les diverses opérations inhérentes au traitement d'une application peuvent être supportées soit par les unités fonctionnelles UF1 et UF3 qui implémentent chacune un additionneur et un multiplieur, soit par les UF2 et UF4 qui sont des ALU proposant plusieurs fonctionnalités (addition, soustraction, valeur absolue, comparaisons, AND, OR, XOR, etc.). Par conséquent, dans MOREA, le motif SAD est mis en œuvre par l'intermédiaire de deux UF2 et d'une UF4, et occupe donc deux *clusters* telle qu'illustrée sur la figure 3.9

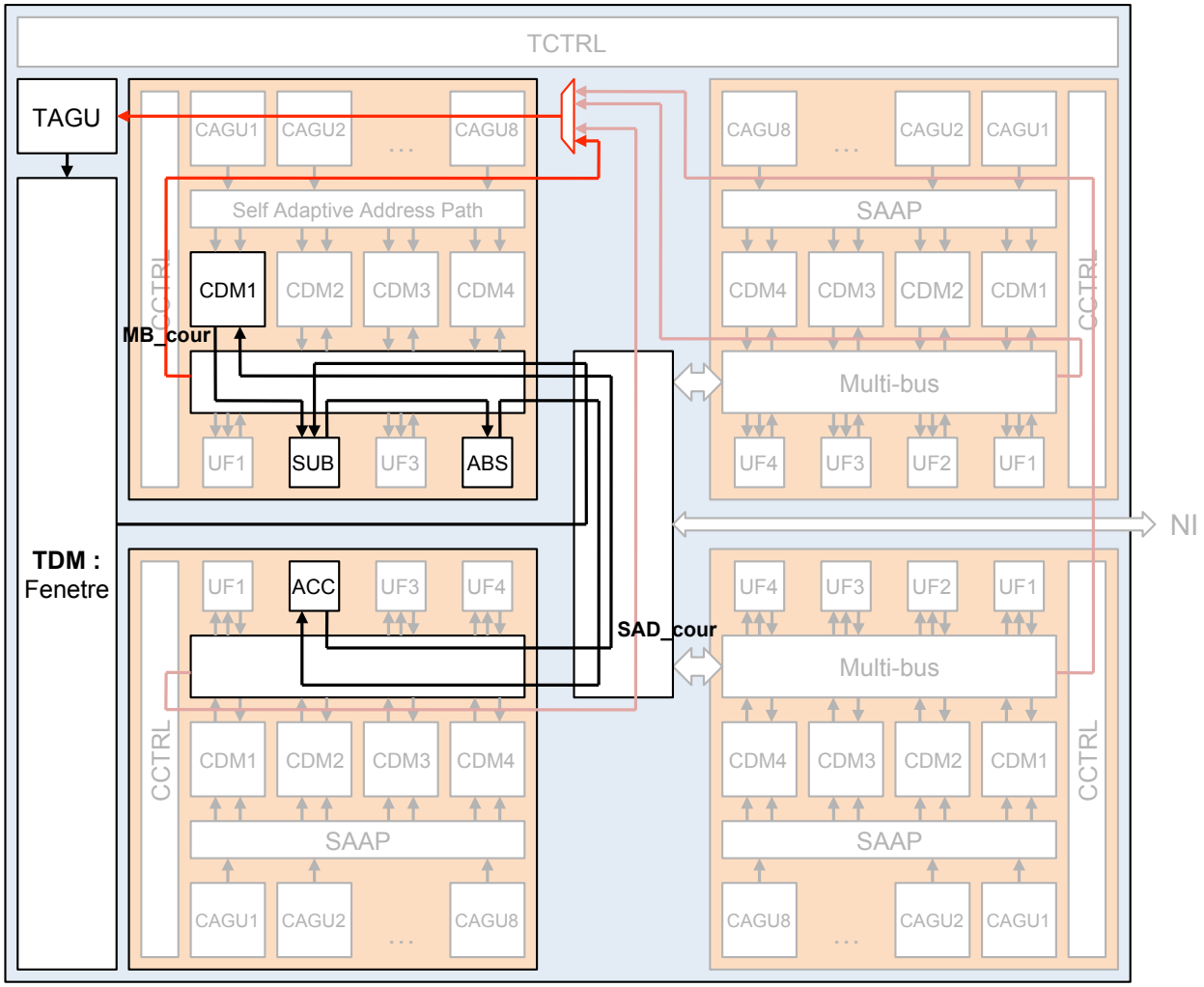


Fig. 3.9: *Mise en œuvre du motif SAD sur une tuile de MOREA. En rouge, est illustrée la mise en œuvre de la dépendance de données exprimée par la séquence d'adresses relative à la lecture des macroblocs de référence candidats. Dans ce cas, une liaison est créée entre le chemin de données de la tuile, en l'occurrence la sortie du multi-bus du cluster 1, et le générateur d'adresses de la mémoire de tuile (TAGU).*

Au cours du calcul, la tâche d'estimation de mouvement consomme des données issues du tampon contenant le macrobloc courant et de celui mémorisant la fenêtre de recherche. Le macrobloc courant est constitué de 256 pixels codés sur 8 bits. Dans un *cluster*, une mémoire simple double-port peut contenir jusqu'à 512 mots de 32 bits. Ainsi, le tampon t_1 est matérialisé par les 256 premières adresses de la mémoire CDM_1 du *cluster* 1 (tableau 3.3). D'autre part, cette tâche parcourt éga-

lement une fenêtre de dimensions 30×30 pixels. Le tampon t_2 correspondant est implémenté par la mémoire de tuile TDM. Finalement, grâce à la structure du réseau d'interconnexions de la tuile, la mémoire de tuile alimente directement les unités fonctionnelles. La « hiérarchie » mémoire de MOREA est donc, dans ce cas de figure, semblable à une structure à un seul niveau et n'impose pas de mouvements de données intra-mémoire.

Adresse	**TDM**	**Cluster 1**			
		CDM1	**CDM2**	**CDM3**	**CDM4**
0	Fenetre[0][0]	MB_cour[0][0]	SAD_min		p
1	Fenetre[0][1]	MB_cour[0][1]	-p		vm.x
2	...	...	delta.x		vm.y
3			delta.y		tmp.x
4					tmp.y
5					tmp.x + delta.x
6					tmp.y +delta.y
...					
255		MB_cour[15][15]			
256		SAD_cour			
...					
899	Fenetre[29][29]				

Tab. 3.3: *Allocation mémoire pour la tâche d'estimation de mouvement.*

Pendant le calcul de la valeur du critère de ressemblance, de part la nature flot de données de ce traitement, son temps d'exécution est déterminé par le temps de calcul des adresses. Plus précisément, il dépend du temps de génération de la séquence correspondant à la lecture de la fenêtre de recherche. En effet, cette séquence d'adresses est affine et présente notamment une dépendance de données. Cette donnée correspond à la position du macrobloc de référence candidat. Dans MOREA, cette dépendance est mise en œuvre en configurant le chemin de données afin notamment de créer une connexion entre ce dernier et le générateur d'adresses de la mémoire de tuile (figure 3.9). Celui-ci mémorise alors dans sa file de registres les données en question via l'exécution d'instructions appropriées.

À ce propos, la figure 3.10 présente le code extrait du programme du générateur d'adresses de la mémoire de tuile, avec unité reconfigurable et sans unité reconfigurable. Ces deux codes assembleur sont la traduction du cœur de nid de boucles principal du code C de l'estimation de mouvement en annexe F. Pour une meilleure lisibilité, le code correspondant est présenté sur la figure 3.11.

Ainsi, les deux codes assembleur permettent de générer les séquences d'adresses associées à la lecture de la variable *Fenetre[i][j]*. D'après le tableau 3.3, l'élément de coordonnées (i, j) de cette variable est mémorisé à l'adresse $g_{\text{Fenetre}}(i, j)$:

```
1         LOAD RA0, x0001 ; RA0 = 1
2         LOAD RA1, x0010 ; RA1 = 16
3         LOAD RA2, x001E ; RA2 = 30
4         LOAD RA3, x00D9 ; RA3 = 217
5         GET RA4 ; RA4 = tmp.x + delta.x
6         ADD RA3, RA3, RA4
7         GET RA4 ; RA4 = tmp.y + delta.y
8         ASH RA5, RA4, +4
9         SUB RA4, RA5, RA4
10        ASH RA4, RA4, +2
11        ADD RA3, RA3, RA4
12        LOAD RA4, x0000 ; indice k = 0
13  etq1: LOAD RA5, x0000 ; indice l = 0
14  etq2: ADD RA6, RA3, RA5
15        OUT R, RA6
16        ADD RA5, RA5, RA6 ; l++
17        LOAD RA6, x0010
18        SUB RA6, RA1, RA5 ; l < 16 ?
19        BNZ etq2
20        ADD RA3, RA3, RA2
21        ADD RA4, RA4, RA0 ; k++
22        SUB RA5, RA4, RA4 ; k < 16 ?
23        BNZ etq1
24        <suite>
```

(a) TAGU sans RU

```
1         LOAD RA0, x00D9 ; RA0 = 217
2         CONF RX6, RA0
3         GET RA1 ; RA1 = tmp.x + delta.x
4         ADD RA0, RA0, RA1
5         GET RA1 ; RA1 = tmp.y + delta.y
6         ASH RA2, RA1, +4
7         SUB RA1, RA2, RA1
8         ASH RA1, RA1, +2
9         ADD RA0, RA0, RA0
10        CONF RX0, RA0
11        LOAD RA1, x000F ; RA1 = 15
12        ADD RA0, RA0, RA1
13        CONF RX2, RA0
14        LOAD RA0, x0001
15        CONF RX1, RA0
16        LOAD RA0, x001E ; RA0 = 30
17        CONF RX3, RA0
18        CONF RX4, RA0
19        ROP R, x00FF
20        <suite>
21
22
23
24
```

(b) TAGU avec RU

Fig. 3.10: *Code extrait du programme du générateur d'adresses de la mémoire de tuile, sans unité reconfigurable (a) et avec unité reconfigurable (b).*

$$g_{\text{Fenetre}}(i,j) = 30 \cdot i + j \tag{3.6}$$

Dès lors, les motifs d'adresses correspondants à générer sont :

$$\text{AE}_{\text{Fenetre}}(k,l) = g_{\text{Fenetre}}(7 + \text{tmp}.y + \text{delta}.y + k, 7 + \text{tmp}.x + \text{delta}.x + l) \tag{3.7}$$

soit :

$$\text{AE}_{\text{Fenetre}}(k,l) = 217 + (\text{tmp}.x + \text{delta}.x) + 30.(\text{tmp}.y + \text{delta}.y) + 30.k + l \tag{3.8}$$

La séquence $\text{AE}_{\text{Fenetre}}(k,l)$ est donc de la forme $C_0 + C_1.k + C_2.l$, avec $C_0 = 217 + (\text{tmp}.x + \text{delta}.x) + 30.(\text{tmp}.y + \text{delta}.y)$, $C_1 = 30$ et $C_2 = 1$. Il s'agit donc bien d'une séquence de type affine où la constante C_0 dépend des données $(\text{tmp}.x + \text{delta}.x)$ et $(\text{tmp}.y + \text{delta}.y)$. Ces dernières sont mémorisées dans la mémoire CDM_4 du *cluster* 1 et elles sont acquises par l'AGU de la mémoire de tuile en établissant, d'une part, un chemin entre l'interconnexion du *cluster* 1 et l'AGU en question (cf. figure 3.9), et, d'autre part au niveau de l'AGU, en utilisant l'instruction <GET RAn> permettant de stocker ces deux données dans la file de registres.

À propos de l'AGU disposant d'une unité reconfigurable (RU[3], la configuration de cette dernière est déterminée à partir de la fonction $\text{AE}_{\text{Fenetre}}(k,l)$. Précisément, en modélisant les indices de boucles

[3]RU : *Reconfigurable Unit*

```
1  /* calcul du critère de ressemblance pour chaque macrobloc de référence */
2  SAD_cour = 0;
3  for(k=0; k<16; k++){
4    for(l=0; l<16; l++){
5      SAD_cour += abs(MB_cour[k][l]-Fenetre[7+tmp.y+delta.y+k][7+tmp.x+delta.x+l]);
6    }
7  }
```

FIG. 3.11: *Code C du cœur du nid de boucles principal de la tâche d'estimation de mouvement.*

k et l par le triplet (0, 1, 15), où 0 est la valeur initiale de l'indice, 1 son pas d'incrémentation et 15 sa valeur finale, on obtient alors les paramètre des registres de configuration de l'unité :

$$\begin{cases} \text{Floor} = \text{AE}_{\text{Fenetre}}(0,0) = 217 + (\text{tmp}.x + \text{delta}.x) + 30.(\text{tmp}.y + \text{delta}.y) \\ \text{Address Step} = \text{AE}_{\text{Fenetre}}(k, l+1) - \text{AE}_{\text{Fenetre}}(k, l) = 1 \\ \text{Base Step} = \text{AE}_{\text{Fenetre}}(k+1, 0) - \text{AE}_{\text{Fenetre}}(k, 0) = 30 \\ \text{Limit} = \text{AE}_{\text{Fenetre}}(0, 15) = 217 + (\text{tmp}.x + \text{delta}.x) + 30.(\text{tmp}.y + \text{delta}.y) + 15 \\ \text{Limit Step} = \text{AE}_{\text{Fenetre}}(k+1, 15) - \text{AE}_{\text{Fenetre}}(k, 15) = 30 \end{cases} \tag{3.9}$$

Le tableau 3.4 compare du point de vue de la surface, des performances et de la consommation d'énergie deux solutions pour la génération d'adresses, dont l'une intègre une unité reconfigurable, exécutant respectivement les codes de la figure 3.10. Ainsi, la présence d'un accélérateur matériel au sein de l'AGU augmente sa surface de 12%. Toutefois, elle permet de limiter la taille du programme de génération d'adresses de 17% dans le cas présent. Ce gain peut dès lors autoriser l'utilisation d'une mémoire d'instructions de moindre capacité et donc contre-balancer l'accroissement en surface dûe à l'intégration de cette unité spécialisée. Par ailleurs, l'exploitation de cette dernière réduit drastiquement de 83% le temps dévolu à la génération de la séquence d'adresses. De plus, cela accroît la puissance de calcul de l'AGU, exprimée en Millions d'Adresses générées Par Seconde, d'un facteur $\times 6$. Cette accélération peut avoir notamment un effet bénéfique sur la consommation de l'ensemble de la tuile. En effet, via la génération d'adresses à un débit élevé, les unités fonctionnelles de MOREA, entre autres, dissipent en majorité de l'énergie dynamique au détriment de celle statique et ce, de part leur activité accrue. En prenant en compte l'augmentation de la consommation statique dans les technologies submicroniques, l'utilisation de cette solution de génération d'adresses peut offrir un gain significatif en consommation. À ce propos, si l'on résonne au niveau de l'AGU, l'intégration d'une unité reconfigurable atténue la puissance dissipée de 69%. Cet état de fait s'explique notamment par l'absence d'étapes de lecture et de décodage d'instructions, et donc d'accès à la mémoire d'instructions, au cours du calcul des adresses. En outre, la réduction conjointe du temps d'exécution de la tâche permet de réduire de 96% l'énergie dissipée (2.48 nJ contre 56.99 nJ).

Suite à ce premier traitement d'estimation de mouvement, le résultat produit est analysé et conditionne la mise à jour de la valeur du vecteur de mouvement. Plus précisément, la valeur calculée est comparée à une valeur minimale et le résultat de cette comparaison engendre ou pas une reconfiguration du chemin de données du *cluster* 1. L'opération de comparaison est menée par l'UF2 qui génère un drapeau indiquant le résultat de manière succinte. Ce signal est ensuite exploité au

	$N_{\text{instructions}}$	**Surface** (μm^2)	N_{cycles}	N_{MAPS}	$\mathbf{P_{AGU}}$ **(mW)**
TAGU w RU	19	71 714	274	180	1.74
TAGU w/o RU	23	63 788	1 628	30	5.6
Gain	*17%*	*-12%*	*83%*	*500%*	*69%*

Tab. 3.4: *Comparaison en surface, performances et consommation de deux structures de génération d'adresses. Ces deux solutions, dont l'une intègre une unité reconfigurable (TAGU w RU), exécutent respectivement les codes de la figure 3.10. D'autre part, outre le nombre de cycles consommés pour le traitement de cette tâche, la performance de ces deux architectures est exprimée en Millions d'Adresses générées Par Seconde (MAPS). Cette métrique est calculée en divisant le nombre d'adresses produites, soit 256 dans le cas présent, par le temps d'exécution du programme en supposant une fréquence de fonctionnement de* 192 MHz.

niveau, entre autres, du contrôleur du *cluster* 1 qui, en fonction de sa valeur, détermine la suite des opérations de configuration. Au final, le vecteur produit par cette première tâche est mémorisé dans la mémoire CDM_4 du *cluster* 1 (tableau 3.3).

3.2.2.2 Description de l'implémentation de la compensation de mouvement

La compensation de mouvement consiste à calculer l'erreur d'estimation du macrobloc courant à partir du macrobloc de référence élu. Il s'agit donc de soustraire aux valeurs des pixels du macrobloc courant, celles des pixels de l'imagette de référence élue. Cette tâche est effectuée par le *cluster* 1 et, plus précisément, par son UF2 configurée pour réaliser une soustraction (figure 3.12). Les données traitées proviennent alors de la première mémoire CDM_1 du *cluster* 1 (macrobloc courant) et de la mémoire de tuile (macrobloc de référence). D'autre part, la séquence d'adresses relative à la lecture du macrobloc de référence dépend de la valeur du vecteur de mouvement calculée lors de la tâche précédente. De ce fait, ces données (vm.x et vm.y) sont transmises au TAGU en établissant un chemin entre ce dernier et la quatrième mémoire CDM_4 du *cluster* 1.

3.2.2.3 Description de l'implémentation de la DCT 2D

Tel qu'évoqué précédemment, la réalisation d'une DCT 2D est similaire à la mise en œuvre de deux DCT 1D successives. Dès lors, son implémentation consiste à mettre en œuvre le graphe de la figure 3.13 sur le chemin de données de la tuile dont il occupe trois *clusters* (figure 3.13). Ainsi, le traitement de la première DCT 1D impose, pour chaque bloc de 4×4 éléments transformé, de lire simultanément les quatre valeurs appartenant à une même ligne. Pour ce faire, la configuration des générateurs d'adresses pour la tâche de compensation de mouvement a été définie telle que ces quatre éléments soit mémorisés à la même adresse mais dans quatre mémoires différentes du *cluster* 1.

Ensuite, pour le traitement de la deuxième DCT 1D, ce sont les quatre valeurs appartenant à une même colonne qui doivent être accédées en parallèle. Dans cette optique, pour chaque bloc intermédiaire produit par la première DCT 1D, les éléments d'une même ligne sont placés dans des mémoires différentes et à des adresses différentes aussi. Concrètement, pour la première ligne, le premier élément est mémorisé dans la mémoire CDM1 à l'adresse i, le deuxième élément dans la mémoire CDM2 à l'adresse $i+1$, etc., pour la deuxième ligne, le premier élément est mémorisé dans

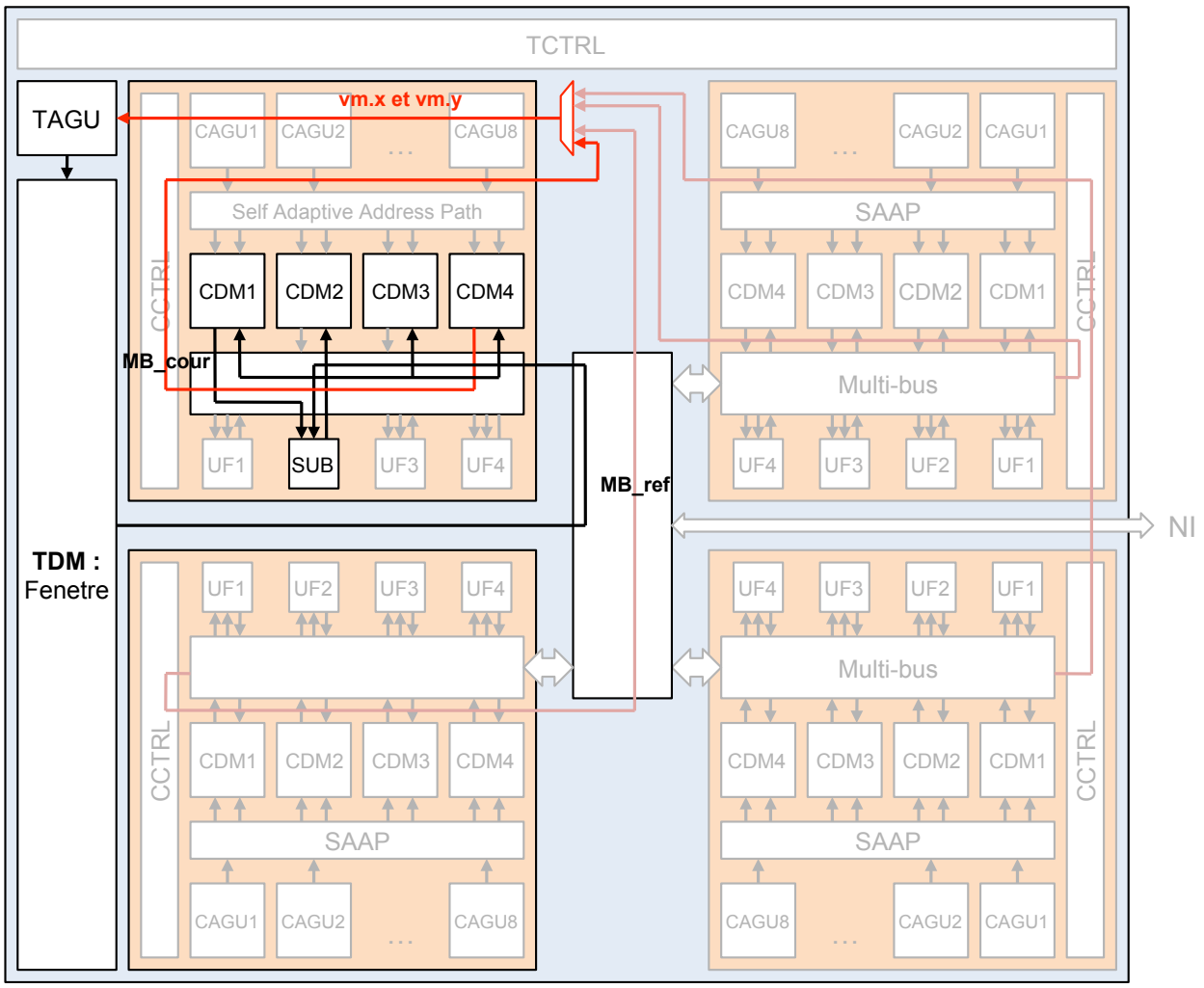


Fig. 3.12: *Implémentation de la tâche de compensation de mouvement sur une tuile de MOREA.*

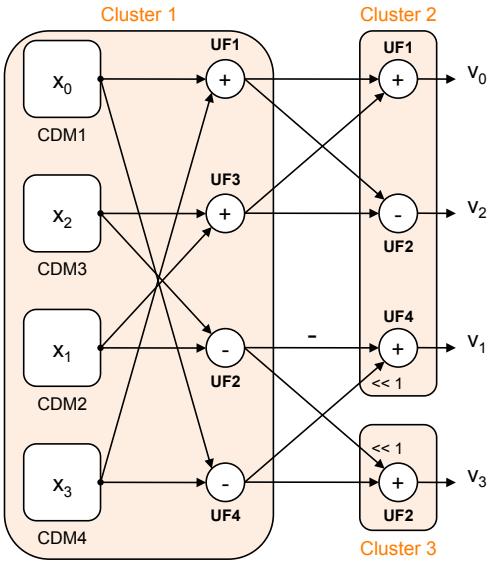


Fig. 3.13: *Mise en œuvre du graphe flot de données de la DCT 1D.*

la mémoire CDM2 à l'adresse i, le deuxième élément dans la mémoire CDM3 à l'adresse $i+1$, etc. La mise en œuvre de cette stratégie impose notamment une reconfiguration fréquente de l'interface mémoire du *cluster* 1 mais qui n'impacte en rien le temps de calcul de cette tâche, ce dernier étant dominé par le temps de génération des différentes adresses.

3.2.2.4 Description de l'enchaînement des tâches de l'application d'encodage vidéo H.264/MPEG-4 AVC

Dans le cadre de la validation de la tuile de MOREA, une application de compression vidéo y a été implémentée. Elle est composée de trois tâches successives (l'estimation de mouvement, la compensation de mouvement et la DCT 2D) dont la mise en œuvre sur une tuile a été présentée séparément dans les paragraphes précédents. Aussi, dans cette dernière partie, nous allons discuter de l'enchaînement de ces tâches et de leur synchronisation.

De manière globale, le contrôleur de tuile est responsable de l'ordonnancement des tâches qui s'exécutent sur les *clusters* de la tuile correspondante. Pour ce faire, deux directives appartenant à son jeu d'instructions sont exploitées :

- l'instruction RUN qui permet d'activer un ou plusieurs contrôleurs de *cluster*, et également le générateur d'adresses de la mémoire de tuile, et donc de lancer l'exécution des tâches ;
- l'instruction WAIT qui, utilisée avec l'option FLAG, permet au TCTRL d'attendre l'activation des drapeaux *fin* des CCTRL, c'est-à-dire d'attendre la terminaison de la ou des tâches en cours d'exécution. L'usage de l'option FLAG est notamment utile dans le cas où une tâche exhibe une dépendance de données et dont la durée ne peut dès lors être déterminée à la compilation. Dans la situation inverse, il peut être possible de préciser un certain nombre de cycles d'attente (WAIT $<N_{\text{cycles}}>$).

Dans le cas de l'application de compression vidéo, suite aux chargements des données à traiter dans les mémoires de la tuile, c'est-à-dire entre autres la fenêtre de recherche dans la mémoire de la tuile et le macrobloc courant dans une mémoire du *cluster* 1, le TCTRL place la première tâche à exécuter, soit celle d'estimation de mouvement. Puisque cette dernière occupe les *clusters* 1 et 2, cette opération de placement consiste à programmer, entre autres, les CCTRL 1 et 2 ainsi que la TAGU qui contrôle la lecture de la fenêtre de recherche. Au niveau du TCTRL, cette phase se résume notamment à exécuter l'instruction NOC qui permet de configurer l'interface de la tuile avec le réseau NoC et d'établir un chemin de communications entre la tuile et la mémoire de configuration système. Puis, l'interface accède en écriture à l'espace mémoire de configuration de la tuile et remplit donc la mémoire d'instructions de ses unités programmables. Pendant ce temps, le TCTRL attend, grâce à la directive WAIT FLAG <masque>, l'activation d'un signal par l'interface NoC lui indiquant la fin de la programmation des différents CCTRL et AGU. À la suite de quoi, le contrôleur de tuile lance l'exécution de l'estimation de mouvement en activant les CCTRL 1 et 2, et la TAGU. L'affirmation du drapeau *fin* de ces unités précise la terminaison de cette tâche.

La deuxième tâche de l'application la compensation de mouvement. Elle occupe un seul *cluster*, en l'occurrence le premier, et nécessite la programmation notamment du CCTRL1 et du TAGU afin

de permettre la lecture du macrobloc de référence élu. Le lancement de l'exécution de cette tâche est réalisée par le TCTRL qui active donc le CCTRL1 et la TAGU. L'affirmation du drapeau *fin* de ces unités indique dès lors la terminaison de cette tâche. Par la suite, la troisième et dernière tâche de la DCT 2D occupe trois *clusters* et impose donc la porgrammation, entre autres, des CCTRL 1, 2 et 3.

3.3 Synthèse

Ce troisième et dernier chapitre a permis, en premier lieu, de caractériser MOREA. Précisément, nous avons estimé la surface, les performances et la consommation d'un *cluster* et d'une tuile. Dès lors, il nous a été possible notamment d'évaluer l'intérêt de notre concept de hiérarchie mémoire reconfigurable. Ainsi, comparativement à une solution classique telle que celle mise en œuvre dans le processeur DART, notre structure de mémorisation améliore le coût en temps et en consommation des communications inter-tâches en éliminant les mouvements de données intra-mémoire.

Ensuite, dans un deuxième temps, nous avons validé notre proposition architecturale, c'est-à-dire que nous avons démontré qu'elle était capable de supporter les différents types de traitement observés dans les applications multimédia. Pour ce faire, nous y avons implémenté les fonctions critiques d'un encodeur vidéo respectant la norme H.264/MPEG-4 AVC. Celles-ci induisent en effet, outre des phases flot de données, des traitements flot de contrôle ainsi que des séquences d'adresses dépendantes des données. Dans ce dernier cas de figure, nous avons pu évaluer la qualité de notre structure de génération d'adresses en la comparant à une approche programmable simple. Ainsi, nous avons observé que l'intégration d'un accélérateur matériel pour le traitement des séquences d'adresses régulières, si elle augmentait légèrement la surface de l'AGU, permettait de minimiser la taille du code, les temps d'exécution et la consommation d'énergie.

Conclusion générale et perspectives

Conclusion générale

Aujourd'hui, les applications embarquées proposent de plus en plus de fonctionnalitées diverses et variées auxquelles s'adjoignent des contraintes de surface, de performance et de consommation. Pour répondre à ces défis, l'éclosion des architectures reconfigurables offre de nouveaux compromis dans le domaine de la conception électronique en matière de surface, d'efficacité énergétique et de flexibilité par rapport aux processeurs programmables et aux composants dédiés. Jusqu'à présent, l'étude de la mise en œuvre de la reconfiguration matérielle a permis d'optimiser, entre autre, la performance et la consommation des ressources opératives et de l'interconnexion. La mémoire, quant à elle, est bien souvent conçue selon une approche statique et hiérarchisée afin de lever le verrou du *memory-processor gap*. Or, dans les systèmes sur silicium, les temps d'accès et la consommation des ressources de stockage dominent la performance et la dissipation énergétique du système. De surcroît, les applications de traitement de l'information induisent des motifs d'accès aux données multiples qui imposent une flexibilisation de la structure mémoire.

Dans ce contexte, plusieurs projets de recherche ont exploré l'application de la reconfigurabilité au niveau de la hiérarchie mémoire, de son interface avec les unités de calcul et de sa structure de génération d'adresses. Ainsi, la reconfiguration de la mémoire aboutit dans la plupart des cas à la possibilité de modifier sa fonctionnalité (par exemple le passage d'une mémoire à accès aléatoire à une structure FIFO voire une LUT) mais pas à l'opportunité d'adapter sa hiérarchie aux caractéristiques des traitements à réaliser. Concernant l'interface mémoire, les solutions proposées sont extrêmement flexibles et reposent sur des interconnexions globales de type *crossbar* ou multibus permettant à toute ressource de calcul d'accéder à n'importe quel banc. Enfin, les unités de génération d'adresses sont développées dans le but de supporter des séquences exclusivement déterministes et ne présentant donc pas de dépendances de données et de contrôle. De manière générale, le contrôle des architectures reconfigurables ne convient pas aux applications dynamiques dont le comportement évolue en fonction des données traitées.

Dans le cadre de nos travaux de thèse, nous proposons donc une architecture reconfigurable dynamiquement qui cible les applications multimédia et de télécommunications mobiles. Celle-ci comprend notamment une hiérarchie mémoire flexible dont l'organisation est capable de s'adapter aux besoins des traitements à implémenter. De plus, cette structure de mémorisation intègre des AGU programmables supportant des séquences d'adresses régulières mais aussi irrégulières et

exhibant des dépendances de données et de contrôle. Globalement, l'unité de contrôle de cette architecture a été élaborée de manière à pouvoir implémenter des applications dont le comportement change dépendamment des données manipulées. Le deuxième chapitre de ce mémoire décrit et justifie l'organisation de notre proposition d'architecture reconfigurable.

Dans le deuxième chapitre, l'architecture de MOREA a été présentée. Préalablement, le modèle d'application que nous considérons a été posé. Il est fondé notamment sur le découpage successif d'une application en processus puis en tâches. À partir de ce modèle, nous avons dégagé l'organisation générale de MOREA. Globalement, celle-ci est structurée autour d'un réseau programmable hiérarchisé qui interconnecte l'ensemble des ressources de calcul et de mémorisation disponibles. De ce fait, il permet d'implémenter une grande diversité de chemins de données entre les unités opératives et de stockage tout en minimisant le coût en surface, délai de propagation et consommation de l'interconnexion.

Dans MOREA, les traitements sont supportés, en premier lieu, par des unités fonctionnelles extraites de l'architecture reconfigurable DART. Dès lors, ces dernières sollicitent, par l'entremise du réseau d'interconnexions, une unité mémoire dont la structure a été définie en fonction des besoins exprimés par les applications de traitement du signal et de l'image, en termes notamment de variété des motifs d'accès mémoire et des séquences d'adresses pour l'unité de génération d'adresses, mais également en tenant compte des contraintes de conception inhérentes au domaine de l'embarqué, à savoir les contraintes de surface, de performances et de dissipation énergétique. Finalement, le contrôle de la reconfiguration des ressources de MOREA est réalisé par une unité *ad-hoc* qui est capable, en outre, de supporter le traitement d'applications dynamiques, c'est-à-dire dont le comportement est dépendant des données traitées par l'application.

Le troisième et dernier chapitre a permis, en premier lieu, de caractériser MOREA. Précisément, nous avons estimé la surface, les performances et la consommation d'un *cluster* et d'une tuile. Dès lors, il nous a été possible notamment d'évaluer l'intérêt de notre concept de hiérarchie mémoire reconfigurable. Ainsi, comparativement à une solution classique telle que celle mise en œuvre dans le processeur DART, notre structure de mémorisation améliore le coût en temps et en consommation des communications inter-tâches en éliminant les mouvements de données intra-mémoire.

Ensuite, dans un deuxième temps, nous avons validé notre proposition architecturale, c'est-à-dire que nous avons démontré qu'elle était capable de supporter les différents types de traitement observés dans les applications multimédia. Pour ce faire, nous y avons implémenté les fonctions critiques d'un encodeur vidéo respectant la norme H.264/MPEG-4 AVC. Celles-ci induisent en effet, outre des phases flot de données, des traitements flot de contrôle ainsi que des séquences d'adresses dépendantes des données. Dans ce dernier cas de figure, nous avons pu évaluer la qualité de notre structure de génération d'adresses en la comparant à une approche programmable simple. Ainsi, nous avons observé que l'intégration d'un accélérateur matériel pour le traitement des séquences d'adresses régulières, si elle augmentait légèrement la surface de l'AGU, permettait de minimiser la taille du code, les temps d'exécution et la consommation d'énergie.

Perspectives

Au cours du chapitre précédent, nous avons pu observer l'impact déterminant des mécanismes de génération d'adresses sur les temps d'exécution de MOREA. Si l'architecture proposée dans le cadre de cette thèse permet d'obtenir un niveau de performance élevé, elle reste cependant perfectible. Ceci est notamment vrai dans le cas du calcul de motifs nécessitant deux phases de configuration ou plus de son unité reconfigurable, tel le motif zig-zag dans le domaine de la compression JPEG par exemple. Dans cet exemple, il est en effet nécessaire de reconfigurer l'unité au cours du calcul de la séquence d'adresses. Une optimisation de l'AGU pourrait être de permettre la définition de deux ou plusieurs contextes de configuration préalablement à la génération des adresses. Au cours de la phase de traitement, l'unité reconfigurable basculerait alors automatiquement d'un contexte au suivant, n'engendrant donc pas d'interruption lors du calcul du motif considéré.

Par ailleurs, le débit élevé des adresses produites par l'AGU permet de limiter les cycles d'inactivité des autres ressources de MOREA (mémoires, unités fonctionnelles, etc.) et donc leur consommation statique, consommation qui ne cesse de croître avec l'émergence des technologies submicroniques. Dans ce contexte, une approche complémentaire pour maîtriser la dissipation énergétique, entre autres, des unités de stockage pourrait être l'introduction de la technique DVFS[4] [54]. Celle-ci permettrait de moduler conjointement la tension d'alimentation et les temps d'accès des mémoires SRAM en fonction des contraintes applicatives. Par exemple, si ces dernières ne nécessitent pas un niveau de performances important, lors d'une période de faible activité, il serait alors envisageable d'abaisser leur tension d'alimentation afin d'augmenter leurs temps d'accès, mais surtout de réduire leur puissance dynamique et statique. Toutefois, cette solution ne semble pas être compatible avec des technologies SRAM 65 nm ou inférieures. En effet, dans ce cas, un abaissement de la tension d'alimentation de 10% entraîne une perte définitive des données mémorisées, limitant de facto l'intérêt de cette technique. Des travaux menés au sein de l'équipe CAIRN visent ainsi, d'une part, à étudier la faisabilité de cette approche et, d'autre part, à quantifier les gains en consommation d'énergie qu'elle pourrait offrir.

À l'heure actuelle, dans le domaine de l'embarqué, la tendance est à la conception d'architectures multi-cœurs. Celles-ci sont notamment constituées de plusieurs cœurs de traitement intégrant des ressources de calcul et de mémorisation, et accédant de manière concurrente à une mémoire partagée. Dans ces solutions, les échanges de données inter-cœurs s'effectuent par l'intermédiaire d'opérations d'écriture et de lecture dans la mémoire partagée qui sont coûteuses en termes de temps et de consommation. Pour limiter ces mouvements de données, une solution semblable à celle mise en œuvre dans une tuile de MOREA pourrait être l'introduction d'une interconnexion flexible entre les différents cœurs de calcul.

Finalement, l'intérêt de la solution que nous proposons est fortement conditionnée par la disponibilité d'un flot de compilation permettant de l'utiliser. Or, MOREA étant une structure parallèle, l'extraction du parallélisme de tâches, d'opérations ou de données à partir d'une description de

[4]DVFS : *Dynamic Voltage and Frequency Scaling*

l'application dans un langage de haut niveau est une opération extrêmement complexe. Au sein de l'équipe CAIRN, des travaux visant à développer des briques de compilation pour extraire différents grains de parallélisme sont en cours. Ces travaux interviennent dans le cadre de la mise en œuvre d'un flot de compilation dynamique (GECOS) qui pourrait être utilisé dans un future proche afin de définir un flot de compilation pour MOREA.

Bibliographie

[1] John L. Hennessy, David A. Patterson, David Goldberg, and Krste Asanovic. *Computer Acrhitecture : A Quantitative Approach.* Morgan Kaufmann Publishers, third edition, 2003.

[2] Pam Tufegdzic. CMOS Image Sensors Continue to Gain Visible Ground. Technical report, iSuppli, 2009.

[3] Odilio Vargas. Achieve minimum power consumption in mobile memory subsystems. `http://www.eetasia.com/`, mars 2006.

[4] Erik Jan Marinissen, Betty Prince, Doris Keitel-Schulz, and Yervant Zorian. Challenges in Embedded Memory Design and Test. In *DATE'05 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 722–727, 2005.

[5] Jim Flynn and Brandon Waldo. Power Management in Complex SoC Design. Synopsys White Paper, avril 2005.

[6] Raphaël David, Dominique Lavenier, and Sébastien Pillement. Du micro-processeur au circuit FPGA : une analyse sous l'angle de la reconfiguration. *Technique et Science Informatiques*, 24(4) :395–422, 2005.

[7] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2) :85–95, 2000.

[8] Christophe Bobda and Reiner Hartenstein. *Introduction to Reconfigurable Computing : Architectures, algorithms and applications.* Springer, 2007.

[9] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, 2005.

[10] Sylvain Collange, David Defour, and Arnaud Tisserand. Power Consumption of GPUs from a Software Perspective. In *Computational Science - ICCS 2009 :* 9^{th} *International Conference*, pages 914–923, 2009.

[11] Paul Heysters, Gerard Smit, and Egbert Molenkamp. A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems. *The Journal of Supercomputing*, 26(3) :283–308, 2003.

[12] Raphaël David, Sébastien Pillement, and Olivier Sentieys. *Low-Power Electronics Design*, chapter Energy-Efficient Reconfigurable Processors, pages 20–1–20–15. CRC Press, 2005.

[13] Gilles Sassatelli, Lionel Torres, Pascal Benoit, Thierry Gil, Camille Diou, Gaston Cambon, and Jérôme Galy. Highly Scalable Dynamically Reconfigurable Systolic Ring-Architecture for DSP

Applications. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, pages 553–558, 2002.

[14] Jürgen Becker, Thilo Pionteck, and Manfred Glesner. DReAM : A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications. *Field-Programmable Logic and Applications : The Roadmap to reconfigurable Computing*, 1896 :312–321, 2000.

[15] Hui Zhang, Vandana Prabhu, Varghese George, Marlene Wan, Martin Benes, Arthur Abnous, and Jan M. Rabaey. A 1-V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing. *IEEE Journal of Solid-State Circuits*, 35(11) :1697–1704, 2000.

[16] Katherine Compton and Scott Hauck. Reconfigurable Computing : A Survey of Systems and Software. *ACM Computing Surveys*, 34(2) :171–210, 2002.

[17] Ian Kuon, Russel Tessier, and Jonathan Rose. FPGA Architecture : Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2) :135–253, 2008.

[18] Eric Kusse and Jan Rabaey. Low-Energy Embedded FPGA Structures. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 155–160, 1998.

[19] Patrick Lysaght, Brandon Blodget, Jeff Mason, Jay Young, and Brendan Bridgford. Invited Paper : Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *FPL'06 : International Conference on Field-Programmable Logic and Applications*, pages 1–6, 2006.

[20] Reiner Hartenstein. A Decade of Reconfigurable Computing : A Visionary Retrospective. In *DATE'01 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 642–649, 2001.

[21] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Eliseu M. Chaves Filho. *MorphoSys* : An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5) :465–481, 2000.

[22] Darren C. Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, and Carl Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In *Proceedings* 20^{th} *Anniversary Conference on Advanced Research in VLSI*, pages 23–40, 1999.

[23] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *MICRO 33 : Proceedings of the* 33^{rd} *annual ACM/IEEE International Symposium on Microarchitecture*, pages 245–257, 2000.

[24] Rama Sangireddy, Huesung Kim, and Arun K. Somani. Low-Power High-Performance Reconfigurable Computing Cache Architectures. *IEEE Transactions on Computers*, 53(10) :1274–1290, 2004.

[25] David H. Albonesi. Selective Cache Ways : On-Demand Cache Resource Allocation. In *MICRO 32 : Proceedings of the* 32^{nd} *Annual International Symposium on Microarchitecture*, pages 248–259, 1999.

[26] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-V_{dd} : A Circuit Technique To Reduce Leakage in Deep-Submicron Cache Memories. In *ISLPED'00 :*

Proceedings of the 2000 International Symposium on Low Power Electronics and Design, pages 90–95, 2000.

[27] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache Decay : Exploiting Generational Behavior to Reduce Cache Leakage Power. In *ISCA'01 : Proceedings of the* 28^{th} *Annual International Symposium on Computer Architecture*, pages 240–251, 2001.

[28] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy Caches : Simple Techniques for Reducing Leakage Power. In *ISCA'02 : Proceedings of the* 29^{th} *Annual International Symposium on Computer Architecture*, pages 148–157, 2002.

[29] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A Highly Configurable Cache Architecture for Embedded Systems. In *ISCA'03 : Proceedings of the* 30^{th} *Annual International Symposium on Computer Architecture*, pages 136–146, 2003.

[30] S. K. Tewksbury, K. Devabattini, and V. Gandakota. A Parallel DSP Testbed with a Heterogeneous and Reconfigurable Network Fabric. In *Proceedings Second Annual IEEE International Conference on Innovative Systems in Silicon*, pages 310–322, 1997.

[31] Tobias Bjerregaard and Shankar Mahadevan. A Survey of Research and Practices of Network-on-Chip. *ACM Computing Surveys*, 38(1) :1–51, 2006.

[32] Juha-Pekka Soininen, Antti Pelkonen, and Jussi Roivainen. Configurable Memory Organisation for Communication Applications. In *Proceedings of the Euromicro Symposium on Digital System Design (DSD'02)*, pages 86–93, 2002.

[33] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles. Predicting Multiprocessor Memory Access Patterns with Learning Models. In *Proceedins of the fourteenth International Conference on Machine Learning*, pages 305–312, 1997.

[34] Rengenarayana Lakshminarayanan and Sanjay Rajopadhye. Switched Memory Architectures - Moving Beyond Systolic Arrays. In *ASAP'03 : Proceedings of the Application-Specific Systems, Architectures and Processors*, pages 28–39, 2003.

[35] Dhiraj K. Pradhan and Nirmala R. Kamath. RTRAM : Reconfigurable and Testable Multi-bit RAM Design. In *Proceedings 'New Frontiers in Testing' International Test Conference*, pages 263–278, 1988.

[36] Subhasis Bhattacharjee and Dhiraj K. Pradhan. LPRAM : A Novel Low-Power High-Performance RAM Design With Testability and Scalability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(5) :637–651, 2004.

[37] Guillermo Talavera, Murali Jayapala, Jordi Carrabina, and Francky Catthoor. Address Generation Optimization for Embedded High-Performance Processors : A Survey. *Journal of Signal Processing Systems*, 53(3) :271–284, 2008.

[38] Samar Yazdani, Joël Cambonie, and Bernard Pottier. Reconfigurable Multimedia Accelerator for Mobile Systems. In *Proceedings of the IEEE International SoC Conference*, pages 287–290, 2008.

[39] Samar Syed Yazdani. *Accès concurrents coordonnés aux mémoires partagées pour les accélérateurs multimédia reconfigurables.* PhD thesis, Université de Bretagne Occidentale, 2008.

[40] Vason P. Srini and Jan M. Rabaey. Invited Paper : Reconfigurable Clusters of Memory and Processors Architecture for Stream Processing Systems. In *Proceedings of the* 6^{th} *International Conference on High Performance Computing in Asia Pacific Region (HPC'02)*, 2002.

[41] Vason P. Srini, John Thendean, and Jan M. Rabaey. Reconfigurable Memory Module in the RAMP System for Stream Processing. In *Proceedings of International Symposium on Computer Architecture Workshop*, pages 113–125, 2001.

[42] Eero Aho. *Design and Implementation of Parallel Memory Architectures.* PhD thesis, Tampere University of Technology, Finland, 2006.

[43] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, Willian J. Dally, and Mark Horowitz. Smart Memories : A Modular Reconfigurable Architecture. In *ISCA'00 : Proceedings of the* 27^{th} *International Symposium on Computer Architecture*, pages 161–171, 2000.

[44] Ken Mai, Ron Ho, Elad Alon, Dean Liu, Tounggon Kim, Dinesh Patil, and Mark A. Horowitz. Architecture and Circuit Techniques for a 1.1-GHz 16-kb Reconfigurable Memory in 0.18-μm CMOS. *IEEE Journal of Solid-State Circuits*, 40(1) :261–275, 2005.

[45] Ken Mai. *Design and Analysis of Reconfigurable Memories.* PhD thesis, Stanford University, 2005.

[46] Raphaël David. *Architecture reconfigurable dynamiquement pour applications mobiles.* PhD thesis, Université de Rennes 1, France, 2003.

[47] Benoit Le Guével and Erwan Grâce. Conception VLSI d'un processeur reconfigurable pour des applications faible puissance. 2005.

[48] Erwan Grâce, Daniel Chillet, Raphaël David, and Olivier Sentieys. MOREA : A Memory-Oriented Reconfigurable Embedded Architecture. In *DASIP'08 : Proceedings of the 2008 Conference on Design and Architectures for Signal and Image Processing*, pages 124–131, 2008.

[49] Scott Hauck, Thomas W. Fry, Mattheuw M. Hosler, and Jeffrey P. Kao. The Chimaera Reconfigurable Functional Unit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2) :206–217, 2004.

[50] Michael Herz, Reiner Hartenstein, Miguel Miranda, Erik Brockmeyer, and Francky Catthoor. Memory Addressing Organization for Stream-Based Reconfigurable Computing. In *ICECS'02 : Proceedings of the* 9^{th} *International Conference on Electronics, Circuits and Systems*, pages 813–817, 2002.

[51] Iain E. G. Richardson. *H.264 and MPEG-4 Video Compression.* Wiley, 2003.

[52] T. Koga, K. Iinuma, A. Hirano, Y. Iijima, and T. Ishiguro. Motion-Compensated Interframe Coding for Video Conference. In *NTC'81 : Proceedings of the National Telecommunications Conference*, pages G5.3.1–G5.3.5, 1981.

[53] Henrique S. Malvar, Antti Hallapuro, Marta Karczewicz, and Louis Kerofsky. Low-Complexity Transform and Quantization in H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7) :598–603, 2003.

[54] Daniel Chillet, Raphaël David, Erwan Grâce, and Olivier Sentieys. Structure mémoire reconfigurable : vers une structure de stockage faible consommation. *Technique et Science Informatiques*, 27(1-2) :181–202, 2008.

Glossaire

AGU *Address Generation Unit*, 22, 50

ALU *Arithmetic and Logic Unit*, 15

ASIC *Application-Specific Integrated Circuit*, 17

CGRA *Coarse-Grained Reconfigurable Architecture*, 20

CMOS *Complementary Metal-Oxide-Semiconductor*, 9

CODEC *COder / DECoder pair*, 10

DCT *Discrete Cosine Transform*, 78

DMA *Direct Memory Access*, 37

DSP *Digital Signal Processor*, 11

DVFS *Dynamic Voltage and Frequency Scaling*, 92

ENIAC *Electronic Numerical Integrator and Computer*, 9

FFT *Fast Fourier Transform*, 25

FIFO *First In First Out*, 39

FPGA *Field-Programmable Gate Array*, 20

GPGPU *General-Purpose computing on GPU*, 17

GPS *Global Positioning System*, 9

GPU *Graphics Processing Unit*, 17

IP *Intellectual Property*, 10

LUT *Look-Up Table*, 20

MOPS *Mega Operations Per Second*, 10

MOREA *Memory-Oriented Reconfigurable Embedded Architecture*, 14

MPEG *Motion Picture Experts Group*, 10

NoC *Network-on-Chip*, 30

PDA *Personal Digital Assistant*, 10

RISC *Reduced Instruction Set Computer*, 37

SIMD *Single Instruction Multiple Data*, 16

SMT *Simultaneous MultiThreading*, 16

SoC *System-on-Chip*, 10

SRAM *Static Random Access Memory*, 23

VLIW *Very Long Instruction Word*, 16

Annexe A

Code C de la fonction *bit-reverse*

```
1  int BitReverse(int data, int width){
2
3     int i,j;
4     int output = 0;
5     int masque = 1;
6     int bit;
7
8     if(width % 2 == 0){
9
10       for(i=0; i<width/2; i++){
11          bit = data && masque;
12          for(j=0; j<width-2*i-1; j++){ bit = bit << 1; }
13          output += bit;
14          masque = masque << 1;
15       }
16
17       for(i=width/2; i<width; i++){
18          bit = data && masque;
19          for(j=0; j<2*(i-width/2)+1; j++){ bit = bit << 1; }
20          output += bit;
21          masque = masque << 1;
22       }
23    }
24    else{
25
26       for(i=0; i<(width-1)/2; i++){
27          bit = data && masque;
28          for(j=0; j<width-1-2*i; j++){ bit = bit << 1; }
29          output += bit;
30          masque = masque << 1;
31       }
32
33       bit = data && masque;
34       output += bit;
35       masque = masque << 1;
36
37       for(i=(width-1)/2; i<width; i++){
38          bit = data && masque;
39          for(j=0; j<2*(i-(width-1)/2); j++){ bit = bit << 1; }
40          output += bit;
41          masque = masque << 1;
42       }
43    }
44    return output;
45 }
```

Fig. A.1: *Code C de la fonction bit-reverse. Cette fonction inverse l'ordre des bits d'une donnée* `data` *de taille* `width` *bits.*

Annexe B

Exemple de script pour l'outil Synopsys Design Compiler

```
1  ####################################################
2  # NOM DU FICHIER : predecodeur_N8.tcl              #
3  # OBJET          : synthèse, estimation en surface #
4  #                  et en consommation d'énergie du #
5  #                  prédécodeur d'adresses de la    #
6  #                  mémoire simple double-port      #
7  ####################################################
8
9  analyze -format vhdl primitive/predecodeur.vhd
10 elaborate predecodeur -parameters "8"
11
12 rename_design predecodeur_N8 predecodeur_N8_gate
13 current_design predecodeur_N8_gate
14
15 link
16 check_design
17
18 set PERIOD 2
19 create_clock CLK -period $PERIOD
20 set_clock_uncertainty 0.1 [all_clocks]
21 set_dont_touch_network [all_clocks]
22
23 remove_driving_cell RESET
24 set_drive 0 RESET
25 set_dont_touch_network RESET
26
27 set all_inputs_wo_rst_clk [remove_from_collection [remove_from_collection [all_inputs] "
       CLK"] "RESET"]
28 set_input_delay [expr ($PERIOD/2)] -clock CLK $all_inputs_wo_rst_clk
29 set_driving_cell -library "CORE9GPLL" -lib_cell "IVLL" -pin Z $all_inputs_wo_rst_clk
30
31 set_output_delay [expr ($PERIOD/2)] -clock CLK [all_outputs]
32 set_load [expr [load_of "CORE9GPLL/IVLL/A"] * 4] [all_outputs]
33
34 compile
35
36 check_timing
37 report_timing
38
39 report_area
40
41 set_switching_activity -static_probability 1 RESET
42 set_switching_activity -static_probability 0.5 -toggle_rate 1 -clock CLK
       $all_inputs_wo_rst_clk
43 set_switching_activity -static_probability 0.5 -toggle_rate 1 -clock CLK -select [list
       regs]
44 propagate_switching_activity -effort high
45 report_power -analysis_effort high
46
47 write -format verilog -output "./netlist/predecodeur_N8_gate.v"
48 write_sdf "./sdf/predecodeur_N8_gate.sdf"
49 write_sdc "./sdc/predecodeur_N8_gate.sdc"
50 write_parasitics -output "./spef/predecodeur_N8_gate.spef"
51
52 exit
```

Fig. B.1: *Exemple de script pour l'outil Synopsys Design Compiler.*

Annexe C

Description du jeu d'instructions d'un générateur d'adresses

Code opératoire	Instruction	Syntaxe	Description
1	LOAD	RAn, < opérande >	Donnée immédiate → RAn
2	GET	RAn	Donnée externe → RAn
3	CONF	RXn, RAm	RAm → RXn
4	OUT	{W, R}, RAn	RAn → @
5	ADD	RAx, RAy, RAz	RAy + RAz → RAx
6	SUB	RAx, RAy, RAz	RAy − RAz → RAx
7	AND	RAx, RAy, RAz	RAy · RAz → RAx
8	ASH	RAx, RAy, $\pm d$	*Arithmetic Shift*
9	ROP	{W, R}, (< N_{cycles} >)	*Reconfigurable Operation*
10	BNZ	< étiquette >	*Branch if Not Zero*
11	BCS	< étiquette >	*Branch if Carry Set*
12	BXF	< étiquette >	*Branch if External Flag Set*
13	BRA	< étiquette >	*Branch Always*
14	WAIT	< N_{cycles} >	*Wait State*
15	END		Programme terminé
0	NOP		*No Operation*

Tab. C.1: *Description du jeu d'instructions du générateur d'adresses de MOREA.*

Registre	Nom du signal
RX0	Floor
RX1	Address Step
RX2	Limit
RX3	Base Step
RX4	Limit Step
RX5	Ceil
RX6	Configuration

Tab. C.2: *Liste des registres auxiliaires du générateur d'adresses de MOREA.*

Annexe D

Description du jeu d'instructions du contrôleur de tuile

Code opératoire	**Instruction**	**Syntaxe**	**Description**
1	LOAD	Rn, < opérande >	Donnée immédiate → Rn
2	GET	Rn	Donnée externe → Rn
3	ADD	Rx, Ry, Rz	Ry + Rz → Rx
4	SUB	Rx, Ry, Rz	Ry − Rz → Rx
5	AND	Rx, Ry, Rz	Ry · Rz → Rx
6	ASH	Rx, Ry, $\pm d$	*Arithmetic Shift*
7	NOC	< requête >	
8	CONF	< cible >, < configuration >	
9	RUN	< masque >	
10	BNZ	< étiquette >	*Branch if Not Zero*
11	BNG	< étiquette >	*Branch if Negative*
12	BXF	< étiquette >	*Branch if External Flag Set*
13	BRA	< étiquette >	*Branch Always*
14	WAIT	(FLAG), < N_{cycles} ou masque >	*Wait State*
15	END		Programme terminé
0	NOP		*No Operation*

Tab. D.1: *Description du jeu d'instructions du contrôleur de tuile.*

Annexe E

Description du jeu d'instructions d'un contrôleur de *clusters*

Code opératoire	**Instruction**	**Syntaxe**	**Description**
1	LOAD	Rn, < opérande >	Donnée immédiate → Rn
2	GET	Rn	Donnée externe → Rn
3	ADD	Rx, Ry, Rz	Ry + Rz → Rx
4	SUB	Rx, Ry, Rz	Ry − Rz → Rx
5	AND	Rx, Ry, Rz	Ry · Rz → Rx
6	ASH	Rx, Ry, $\pm d$	*Arithmetic Shift*
8	CSW	< sélection >, < configuration >	Configuration *Software*
8	CHW	< cible >, < configuration >	Configuration *Hardware*
9	RUN	< masque >	
10	BNZ	< étiquette >	*Branch if Not Zero*
11	BNG	< étiquette >	*Branch if Negative*
12	BXF	< étiquette >	*Branch if External Flag Set*
13	BRA	< étiquette >	*Branch Always*
14	WAIT	(FLAG), < N_{cycles} ou masque >	*Wait State*
15	END		Programme terminé
0	NOP		*No Operation*

Tab. E.1: *Description du jeu d'instructions d'un contrôleur de clusters.*

Annexe F

Code C de la fonction d'estimation de mouvement

```
1  void EstimMvt(int MB_cour[][16], int Fenetre[][30], vect *vm){
2
3    int SAD_min = 0;
4    int SAD_cour;
5    int i,j,k,l;
6    int step;
7    int p = 4;
8    vect tmp, delta;
9
10   /* calcul du critère de ressemblance */
11   /* pour le premier macrobloc de référence, i.e le macrobloc central */
12   for(i=0; i<16; i++){
13     for(j=0; j<16; j++){
14       SAD_min += abs(MB_cour[i][j]-Fenetre[7+i][7+j]);
15     }
16   }
17
18   /* initialisation */
19   *vm.x = 0;
20   *vm.y = 0;
21   tmp.x = 0;
22   tmp.y = 0;
23
24   /* pour chaque étape de l'algorithme */
25   for(step=0; step<3; step++){
26     delta.y = -p;
27     for (i=0; i<3; i++){
28       delta.x = -p;
29       for(j=0; j<3; j++){
30
31    if(i!=1 || j!=1){
32
33      /* calcul du critère de ressemblance pour chaque macrobloc de référence */
34      SAD_cour = 0;
35      for(k=0; k<16; k++){
36        for(l=0; l<16; l++){
37          SAD_cour += abs(MB_cour[k][l]-Fenetre[7+tmp.y+delta.y+k][7+tmp.x+delta.x+l]);
38        }
39      }
40
41      /* mise à jour du vecteur de mouvement */
42      if(SAD_cour < SAD_min){
43        SAD_min = SAD_cour;
44        *vm.x = tmp.x + delta.x;
45        *vm.y = tmp.y + delta.y;
46      }
47    }
48    delta.x += p;
49       }
50       delta.y += p;
51     }
52     tmp.x = *vm.x;
53     tmp.y = *vm.y;
54     p = (p >> 1);
55   }
56 }
```

Fig. F.1: *Code C de la fonction d'estimation de mouvement.*